Ark: Offchain Transaction Batching in Bitcoin

***Abstract*—Bitcoin is the cryptocurrency with the largest market capitalisation, but its widespread adoption is fundamentally limited by the scalability constraints of its consensus algorithm, which requires every transaction to be confirmed onchain. To address this, several Layer-2 scalability solutions have been proposed to move payments offchain—most notably, the Lightning Network. However, their deployment remains hindered by cumbersome setup requirements: users must lock funds on-chain to participate and engage in complex auxiliary protocols (e.g., for channel rebalancing, top-ups, and routing). Other solutions, like payment pools, sidechains, commit-chains and rollups cannot be implemented in a non-custodial way on Bitcoin due to its limited scripting capabilities, or require all protocol participants to update the offchain state.**

In this work, we present Ark, the first Bitcoin-compatible commit-chain. Ark enables offchain transactions of virtual UTXOs (VTXOs), through an untrusted operator who aggregates them into succinct onchain commitments. A distinctive feature of Ark is its ease of deployment: users can receive offchain payments without locking any funds beforehand and Ark state updates can be performed only requiring the users involved in that update.

We formally define the Ark protocol and prove its security. During this process, we identified two attacks affecting the testnet implementation, which we responsibly disclosed and proposed fixes for, which have been now integrated into the mainnet implementation. Our experimental evaluation demonstrates that Ark can commit on-chain to batches of arbitrarily many VTXOs with a constant-sized footprint of approximately 200 vB. Cooperative exits add one output per user, while unilateral exits require $\mathcal{O}(\log n)$ transactions of roughly 150 vB per VTXO for a batch of n VTXOs.

1. Introduction

The security and decentralisation guarantees of the Bitcoin consensus protocol [1] come at the expense of severely limited throughput—approximately 10 transactions per second—about three orders of magnitude lower than conventional credit card systems, which can process around 10k transactions per second. To overcome this limitation, a variety of Layer-2 protocols have been proposed, most notably payment channel networks [2], [3], [4], [5], [6], [7], [8] and their extensions (such as payment channel hubs [9], [10], [11], [12], virtual channels [10], [13], [14], [15], and channel factories [16], [17]). However, these approaches share a fundamental drawback: they require a *cumbersome and*, most importantly, *costly setup*. Specifically, users must lock collateral onchain to initially fund their channels as well as to engage later on in complex auxiliary protocols (e.g., for channel rebalancing, channel top-ups, and payment routing). These usability and liquidity challenges have hindered the widespread deployment of such protocols.

In this work, we present Ark, the first Bitcoin-compatible commit-chain. A distinctive strength of Ark lies in its *open architecture*. In contrast to payment channel solutions, users can receive funds without issuing any onchain transaction, thereby lowering entry barriers. At the same time, funds deposited in Ark remain fully fungible: they can be transferred to anyone, Ark user or not, who may subsequently decide to use them within Ark or move them outside—for example, in order to make onchain Bitcoin payments or even to setup a Layer-2 protocol on top of Ark. This design eases onboarding, enhances interoperability, and mitigates liquidity fragmentation.

At its core, Ark introduces the concept of **virtual UTXOs** (VTXOs), which serve as offchain counterparts of standard UTXOs. Ark allows users to transact offchain by drawing on shared liquidity managed by an operator. Each transaction updates the allocation of VTXOs, and the operator periodically commits state updates to the Bitcoin blockchain. Furthermore, Ark features *unilateral exit*: each user may decide to bring their VTXOs onchain without forcing any other Ark participant to do that. Additionally, Ark imposes *minimal online requirements*: only the operator must remain continuously online, while users need to connect periodically to refresh their VTXOs—a process that occurs entirely offchain. Finally, in the presence of a rational operator, an existentially honest signer committee, and assuming at all times there is at least one honest online user, Ark enables *fast finality*: users can spend their VTXOs immediately, without the risk of the transaction being reverted.

1.1. Related work

The comparison between Ark and other Layer-2 protocols is summarised in Table 1 and discussed below. We omit custodial approaches like eCash [18], [19], [20].

Payment channel networks (PCN). PCNs, such as the Lightning Network [2], are networks of 2-party channels, i.e., 2-of-2 multisignature UTXOs, where parties can cooperate to instantly update their channel balances, and where any party can at any time close the channel unilaterally and claim its funds after a dispute period, in which the counterparty should come online to prevent a malicious user from exiting with an old state. PCNs therefore achieve *fast finality*

(transactions are instant and non-revertible) and *unilateral exit* (users can exit their funds without the collaboration of another party). Closing, as well as opening a channel, has an onchain cost and requires to lock funds, which can only be used *within* the network. Ark, on the other hand, is *externally fungible*: it allows users who are not onboarded to receive funds with no onchain footprint. Moreover, PCNs require routing algorithms to find a path with sufficient capacity in the network, rebalancing algorithms to reallocate funds among channels, and possibly onchain top-up transactions. Payment channel hubs [9], [10], [11], [12], virtual channels [10], [13], [14], [15] and state channels [21], [22], [23], which can track arbitrary state, suffer from similar limitations. Channel factories [16], [17] alleviate these issues by creating multiple channels at once, amongst which liquidity can be reallocated. However, one user leaving leads to other users being dragged out as well.

Statechains. Somsen [24] proposed statechains to instantly transfer ownership of a UTXO to arbitrary parties with no onchain activity, and thus achieve fast finality and external fungibility. The owner can claim funds unilaterally after a dispute period. Spark [25] allows splitting the UTXO amongst multiple owners. However, statechains are secure only under the assumption of a trusted operator that cosigns each transfer and deletes its key after each state update. Ark similarly enables operator-assisted transfers, but the operator can be malicious: we just need to assume their rationality for fast finality.

Payment pools. The most developed example is CoinPool [26]. It allows multiple users to share a single UTXO, make instant offchain transfers within this UTXO, and allows unilateral exit at any time by any user. Contrary to Ark, it assumes the introduction of several new Bitcoin opcodes (as of the current, post-Taproot, state of Bitcoin: it is thus not *Bitcoin-compatible*), and has a high interactivity requirement since each state update requires each user's approval.

Sidechains. Sidechains [27], [28], [29] are independent blockchains, able to process transactions to arbitrary users with faster confirmation times (but not necessarily with fast finality), or richer smart contract functionality, running in parallel to the main blockchain, enabling the transfer of assets between them via a two-way peg mechanism. In Bitcoin, this approach has to be custodial, with users sending funds to a trusted committee, requiring the its collaboration to exit again. BitVM [30], [31] could partially remove this trust. Regardless, sidechains reduce their security from that of Bitcoin to that of the sidechain's consensus mechanism.

Commit-chains. Here, an untrusted operator coordinates transactions between its participants (not externally fungible) and periodically commits the state to the blockchain. Most examples of commit-chains [32], [33] assume an expressive blockchain to handle user exits and disputes and are thus not Bitcoin-compatible. Commit-chains achieve delayed finality, i.e., the payment is only finalised once the corresponding commitment is confirmed onchain, and survived the challenge period (in which participants should come online to contest any misbehaviour). Assuming a malicious operator, Ark also achieves delayed finality (but a rational operator enables fast finality). Clique [34] is a commit-chain handling simple payments on Bitcoin. Its original design requires a covenant functionality Bitcoin currently does not have, but could be modified to instead use signatures from all users to update the state. In contrast, Ark only requires signatures of users to perform a state update and is fully Bitcoin-compatible. Contrary to other commit-chains, both Clique and Ark operators fund the onchain commitment with their own liquidity.

Rollups. Rollups [35], [36] offload computation offchain, but post all transaction data onchain. Just like commit-chains, rollups achieve unilateral exit and delayed finality, but are not externally fungible, and they are in general not Bitcoin-compatible. BitVM [30], [31] might in principle be leveraged to design Bitcoin-compatible rollups, but it is very expensive (e.g., dispute require 0.15 BTC in fees and more than 3MB in size). Ark instead focuses on reducing its onchain footprint, having only a small transaction onchain committing to a state update. In contrast to rollups, where the state can be derived from onchain data, each Ark user stores its local state.

TABLE 1: Protocol comparison. Columns: UE: unilateral exit; EF: external fungibility (no need for onboarding to receive funds); FF: fast finality (safely spend funds immediately, D = delayed until onchain commitment, * under rational operator and 1-of-n honest committee and user, otherwise D for Ark); SU: parties required per state update (except unilateral exit) (op. = operator); OL: operator liquidity; BC: Bitcoin-compatible; OR: users should come online each: (DP = Dispute period, OC = Onchain commitment).

	PCN	State-chain	Payment Pool	Side-chain	Commit-chain	Bitcoin Clique	Rollup	Ark
TA		Honest op.		Honest validators				Rational op. only for FF
UE	✓	✓	✓		✓	✓	✓	✓
EF		✓		✓				✓
FF	✓	✓	✓		D	D	D	✓*
SU	channel parties	owner + op.	all	validators	op.	all	op.	op. + involved users
CL		0			0	Clique value	0	Batch value
BC	✓	✓		✓		✓		✓
OR	DP	DP	DP	Never	OC	OC	OC	VTXO lifetime

1.2. Our contributions

The contributions of this work can be summarised as follows:

- We present Ark, the first Bitcoin-compatible Layer-2 scalability protocol, featuring unilateral exit, fast finality, and an open architecture, allowing receivers to join without performing any onchain transaction, and funds to be spent within as well as outside Ark.
- We formalise and prove the security of Ark (VTXO security, user balance security, atomicity, and operator balance

security). In doing so, we identified two attacks on the original Ark protocol, as implemented in the testnet: the hostage attack and spam attack. We solve these attacks by introducing reset transactions, which have been integrated into the mainnet implementation.
- We conduct an experimental evaluation to demonstrate the performance and scalability gains. Specifically, we show that the onchain footprint for a commitment with arbitrarily many VTXOs is constant (197 vB) and for unilateral exit is logarithmic in the number n of VTXOs per commitment ($\lceil \log n \rceil \cdot 150 + 107$ vB), which is a reasonable price to pay to obtain the functionality of a commit-chain in Bitcoin. FInally, we show the time required to produce a commitment is linear in the number of users involved in the commitment. For example, it requires a mere 2.7 seconds for 200 users.

2. Model and Protocol Overview

2.1. Model and Notation

System Model. Ark is a Bitcoin-native transaction batching protocol that lets an arbitrary set of users $\mathcal{P} = \{P_1, P_2, \ldots\}$ transact offchain with a minimal onchain footprint, using an untrusted *Ark operator* O. Users hold virtual balances (represented by *VTXOs*) and issue Ark requests, such as boarding (onramp), transactions, batch swaps, and exits (offramp). Users can moreover claim their funds onchain at any time. O aggregates requests and *batches* them into onchain *commitment transactions*, which are approved by the involved users in signing sessions coordinated by O, and are predominantly funded by O.

Blockchain and UTXO Model. We assume a robust public transaction ledger $\mathcal{A}$ (Bitcoin) that operates in the UTXO model. The ledger $\mathcal{A}$ consists of a sequence of transactions.

Transactions and Script. In the UTXO model, each transaction $\texttt{tx}$ maps a non-empty list of existing unspent transaction outputs (UTXOs) to a list of new UTXOs. A transaction is a tuple $\texttt{tx} = (\texttt{ins}, \texttt{wits}, \texttt{outs})$, where $\texttt{ins} = [\texttt{out}_1, \ldots, \texttt{out}_n]$ references unspent outputs, $\texttt{wits} = [w_1, \ldots, w_n]$ provides unlocking data (witnesses), and $\texttt{outs} = [\texttt{out}^\star_1, \ldots, \texttt{out}^\star_m]$ are new outputs. Each output $\texttt{out} = (\texttt{value}, \texttt{lockScript})$ holds an amount $\texttt{out.value}$, locked by a locking script $\texttt{out.lockScript}$. A transaction is *valid* if every input is unspent, each script accepts its witness, i.e., $\texttt{out}_i.\texttt{lockScript}(w_i) = \texttt{True}$, and if $\sum_{i=1}^{n} \texttt{out}_i.\texttt{value} \geq \sum_{j=1}^{m} \texttt{out}^\star_j.\texttt{value}$.

The locking scripts are expressed in the stack-based Bitcoin scripting language. We use: (i) signature checks $\texttt{chkSig}_{pk}$ that require a valid signature under pk, (ii) k-of-n multisignature checks $\texttt{chkMulSig}_{k,n;pk_1,\ldots,pk_n}$, (iii) absolute timelocks $\texttt{absTlk}(T)$ and relative timelocks $\texttt{relTlk}(t)$. Taproot allows key-path spends with Schnorr signatures or script-path spends by revealing a single committed script together with a Merkle proof. We denote a Taproot output by $\texttt{Taproot}(pk_I; \texttt{scriptPath}_1, \ldots, \texttt{scriptPath}_n)$, where pk_I is the internal key used together with the root of the script path Merkle tree to determine the key path spend public key, and is set to $\texttt{False}$ when the key path is disabled.

Ledger Model. We adopt the blockchain (ledger) model of [37] to which parties can submit transactions. In this model, each participant of the blockchain protocol (miner) M maintains a local view of the ledger $\mathcal{L}^M$; write $\mathcal{L}^M_{-k}$ for $\mathcal{L}^M$ without the last k blocks. We assume our ledger is robust with depth and wait parameters k, u, i.e., it satisfies the following properties (informally): with overwhelming probability, (i) if $\texttt{tx} \in \mathcal{L}^M_{-k}$ for some honest M, it will be reported by the other honest miners at the same position in their ledger views (persistence), and (ii) if $\texttt{tx}$ is given to all honest miners for u consecutive communication rounds, then each honest M will report $\texttt{tx} \in \mathcal{L}^M_{-k}$ (liveness). We use this interface to reason about confirmation and reorgs. We denote for any Ark party (user or O) Q by $\mathcal{L}^Q$ the local view of Q, where Q shares the view of some miner M.

Communication Model. Parties communicate over authenticated channels. The network is synchronous, i.e., all messages are delivered within a known time bound Δ (which is also the network model of most blockchains, including Bitcoin). We assume O is online, while users may be intermittently online. Users performing fast finality payments need to remain online until their funds are spent.

Cryptographic and Script Assumptions. We assume standard cryptographic primitives, i.e., collision-resistant hash functions and EUF-CMA secure signature schemes. Taproot Schnorr signatures are used for key/script paths and for efficient n-of-n multisignature via MuSig2 [38] (aggregate key denoted $\bigoplus_i pk_i$, checked by $\texttt{chkSig}_{\bigoplus_i pk_i}$). Finally, we remark that as script-level covenants are unavailable, spend restrictions can be emulated with n-of-n co-signing. Stronger covenants would reduce interactivity but are not required for correctness.

Threat Model. A PPT adversary may corrupt any subset of parties (including O), possibly adaptively and with full collusion. The adversary controls message scheduling (subject to the synchronous bound) and the mempool, but cannot break cryptography or the ledger's security properties (but with negligible probability). We consider the adversary to be Byzantine, i.e., it may deviate from the protocol arbitrarily. We further provide an extension enabling fast finality, where we assume rationality of the operator (i.e., it can misbehave only if this leads to a financial gain) and existential honesty for (i.e., at least one being honest among) the signing committee and the Ark users.

2.2. Protocol Overview

We use three recurring objects: (i) *VTXOs*: offchain outputs with at least one collaborative spend path (requiring O's signature) and at least one unilateral spend path (user only, after a relative timelock); (ii) *batches*: onchain outputs that compactly commit to a set of VTXOs; and

(iii) *connectors*: onchain outputs committing to multiple *anchors*, outputs that atomically bind offchain updates to the next commitment (cf. §3.3). Figure 1 illustrates the end-to-end flow; we summarise the components below and cross-reference the detailed sections.

- **Running Ark,** §3.1,§3.4. The operator O periodically broadcasts *commitment transactions*, typically funded by its own liquidity. These commitments contain *batches* that compactly commit to the set of newly created VTXOs.
- **Transacting within Ark,** §3.2. Users collaborate with O to build offchain (virtual) transactions that spend existing VTXOs and create new VTXOs with specific scripts.
- **Batch swapping,** §3.3. Users exchange (offchain) old VTXOs for new ones in the next batch by *forfeiting* the old VTXOs (allowing O to claim them if they go onchain). *Connectors* atomically bind the swap to the next commitment.
- **Boarding,** §3.4. A user posts a boarding transaction whose output is spendable either jointly with O or unilaterally after a timelock. Once confirmed, O spends it in the next commitment, giving VTXOs to the user for offchain use.
- **Exiting,** §3.4. To exit, the user forfeits her VTXOs and receives onchain outputs in the next commitment. If O is unresponsive or adversarial, the user exits unilaterally via the batch's script path.

2.3. Security and Scalability Properties

We introduce the security and performance guarantees below; which we informally prove in §5. We introduce the security and performance guarantees below; which we informally prove in §5. Formal statements and proofs are postponed to Appendix F.

The Ark protocol maintains a set of unspent VTXOs, which we can see as offchain state. Intuitively, VTXO Security guarantees that any Ark state can be mapped to a corresponding Bitcoin state (i.e., a set of UTXOs). Ark Atomicity then guarantees that the transitions between these Ark states are well-behaved.

- **VTXO Security.** No unexpired[1] VTXO can be spent without a valid witness (safety). Any VTXO held by an honest user can be unilaterally redeemed onchain before batch expiry (liveness).
- **Ark Atomicity.** For any Ark action (boarding, transaction, batch swap, exit), either all inputs are consumed and all outputs are created, or none is.

From these two properties, we can derive *balance security*, for both the users and the operator.

Finally, with respect to scalability, Ark's onchain footprint is: (i) *execution constant*: the onchain commitment size is constant in the number of VTXOs; (ii) *optimistic exit constant*: with a responsive operator, a user exits with $\mathcal{O}(1)$ onchain transactions; and (iii) *pessimistic exit logarithmic*: in the worst case, a user exits a VTXO with $\mathcal{O}(\log n)$ transactions, where n is the batch's VTXO population.

1. Expired batches of VTXO can be claimed at once by O.

3. Construction

At the core of Ark is an operator O with key pair (sk_O, pk_O) who facilitates transactions of *virtual UTXOs* (VTXOs). VTXOs are offchain outputs that can be spent without leaving an onchain footprint (§3.2). Each VTXO is locked by a Taproot script with two paths: a *collaborative* spend with O, and a *unilateral* spend that requires no interaction with O.

Definition 3.1 (Virtual UTXO / VTXO)**.** *A* virtual UTXO *or* VTXO*, with operator O, is a transaction output* $\texttt{vtxo} := (\texttt{value}, \texttt{vtxoLockScript})$ *where* $\texttt{vtxoLockScript}$ *is a Taproot locking script with: (i) an unspendable key path; (ii) at least one* collaborative *script path, requiring a signature of O and possibly an absolute timelock; and (iii) at least one* unilateral *script path, not requiring O's signature and delayed by a relative timelock t_v (determined by O), ensuring that a VTXO appearing onchain can be spent collaboratively at least t_v blocks before it becomes unilaterally spendable. A VTXO is called* collaboratively *or* unilaterally *spent if the corresponding path is used.*

The simplest VTXO is a *single-signature* output: an amount an Ark user Alice, with key pair (sk_A, pk_A), can spend via a signature under pk_A. Its locking script $\texttt{Taproot}(\texttt{False}; \texttt{chkSig}_{pk_O \oplus pk_A}, \texttt{chkSig}_{pk_A} \wedge \texttt{relTlk}(t_v))$ requires a collaborative witness $w_{collab} = \sigma_{O \oplus A}$ or a unilateral witness $w_{unilat} = \sigma_A$. For simplicity, we assume in this section that VTXOs have a clear "owner" (e.g., Alice or Bob), who can spend them both collaboratively and unilaterally. However, VTXOs may include multiple complex script paths, generalising ownership to anyone providing a valid witness. This is assumed in the rest of the paper.

Throughout the rest of this paper, we adopt two conventions. First, when $\texttt{vtxo} = (\texttt{value}, \texttt{vtxoLockScript})$ appears as an output in a (virtual) transaction, it represents a value $\texttt{value}$ and a Taproot $\texttt{scriptPubKey}$ of the form $\texttt{OP\_1 OP\_PUSHBYTES\_32}\ pk_T$, where pk_T is the tweaked 32-byte public key committing to the key and script paths in $\texttt{vtxoLockScript}$. A blockchain observer only sees pk_T and cannot infer the spending paths. In other contexts (e.g., an Ark user sharing $\texttt{vtxo}$ with O), both $\texttt{value}$ and all spending paths are shared, allowing derivation of pk_T. Second, when requests create new VTXOs, these are initially just value–script pairs, not yet tied to a specific transaction. Context clarifies whether we refer to this pair or to a fully defined VTXO within a batch.

We now describe Ark's core functionality: transaction batching, which consolidates multiple VTXOs into a single onchain output called a *batch*. Users can transact offchain by spending and creating VTXOs, and batch swapping atomically exchanges these new VTXOs for fresh ones in a new batch. This process motivates the design of commitment transactions, the only onchain footprint. We then extend this structure to support Ark user entry and exit.

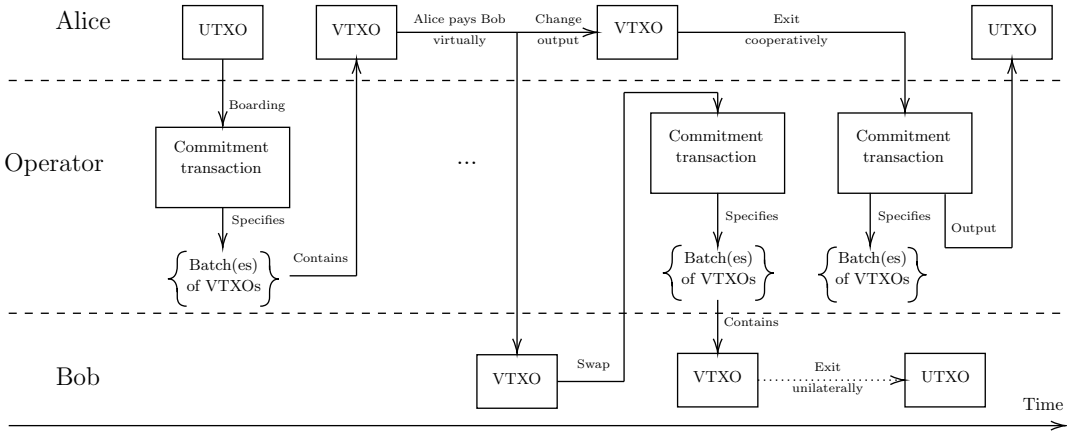

Figure 1: An example flow within the Ark protocol.

3.1. Transaction Batching

As the name suggests, VTXOs are meant to remain virtual. While they can be unilaterally converted into UTXOs via an exit path, this is ideally avoided. Instead, VTXO holders are expected to collaborate with O to claim funds onchain (see §3.4). This allows us to design a structure that minimises onchain footprint while still permitting unilateral exits if needed. This structure, called a *batch*, is a transaction output predominantly funded by O that can be spent either via a *sweep* path (returning funds to O) or an *unroll* path, which splits the output into VTXOs via a *virtual transaction tree (VTXT)*. This tree is made up of virtual transactions: Bitcoin transactions that will optimistically never go onchain, specifying how the batch should be distributed amongst VTXOs.

Definition 3.2 (Virtual transaction tree)**.** *A* virtual transaction tree (VTXT) *is a directed rooted tree, given by the ordered pair $G = (V, A)$, where V is a set of virtual transactions (the* nodes*) with exactly one input, and A is a set of ordered pairs of virtual transactions (the* edges*), such that, for every $v \in V$, there exists a unique $(v_1, v_2) \in A$ such that $v_2.\texttt{ins} \supseteq v_1.\texttt{outs}$. There is also exactly one virtual transaction $r \in V$, called the* root*, such that there are no edges $a \in A$ of the form (v, r), where $v \in V$. Any $\ell \in V$ for which there are no edges of the form (ℓ, v) in A (for $v \in V$) is called a* leaf*. For any $v \in V$, we define $\texttt{path}(v)$ as the sequence $(v_1, \ldots, v_\ell)$, where v_1 is the root, $v_\ell = v$, and $(v_i, v_{i+1}) \in A$ for each $i \in \{1, \ldots, \ell - 1\}$. For any $\texttt{out} \in v.\texttt{outs}$, we interpret $\texttt{path}(\texttt{out})$ as $\texttt{path}(v)$.*

Definition 3.3 (Batch)**.** *A* batch *is a transaction output locked by a Taproot script with an unspendable key path and exactly two script paths: (i) a* sweep *path that allows O to claim the entire output after a certain block height, called the* batch expiry*, and (ii) an* unroll *path that specifies spending according to a VTXT with root spending the full batch, each leaf a VTXO as its only output, and the remaining nodes virtual transactions that have batches as their only outputs.*

Remark 3.4. *The VTXT structure is enforced by a covenant, ensuring the batch can be spent* only *according to the VTXT before expiry. To preserve Bitcoin compatibility, this covenant can be emulated with an n-of-n multisignature (e.g., Musig2), where O coordinates signing sessions with all VTXO holders. Either all holders sign every virtual transaction, or, as in Figure 2, each holder signs only along the path to their VTXO. The latter reduces the number of interactions without compromising safety, since every honest holder must approve the transactions needed to redeem their VTXO, ensuring their batch funds cannot be spent otherwise.*

The purpose of batch expiry is explained in §3.4. For now, note that any VTXO holder who knows the virtual transaction path in the VTXT to their VTXO can broadcast it onchain and exit unilaterally. Figure 2 illustrates this: to redeem v_1, P_1 must publish the transactions with outputs (v_1+v_2, v_3+v_4), (v_1, v_2), and v_1. Only P_1's VTXO appears onchain, but P_1 must pay fees, which may make small VTXOs uneconomical to exit.

3.2. Ark transactions

As discussed earlier, VTXOs are intended to remain offchain, enabling the offchain execution of transactions via *Ark transactions*. These spend one or more VTXOs from a batch through the collaborative path, creating new VTXOs.

To illustrate the Ark transaction mechanism, consider an Ark user Alice who wants to send amount p to Bob. Alice holds a VTXO $\texttt{vtxo}_A$ with value $a > p$ in a batch. She constructs an Ark transaction $\texttt{tx}_A^{\texttt{ark}}$, a virtual transaction with inputs $[\texttt{vtxo}_{i,1}, \ldots, \texttt{vtxo}_{i,n}]$, collaborative witnesses $[w^1_{collab}, \ldots, w^n_{collab}]$, and outputs $[\texttt{vtxo}_{o,1}, \ldots, \texttt{vtxo}_{o,m}]$. In this example, $n = 1$, $\texttt{vtxo}_{1,1} = \texttt{vtxo}_A$, $w^1_{collab} = \sigma_{O \oplus A}$, and the outputs may include: a VTXO $\texttt{vtxo}_B$ with amount p spendable by Bob, a change VTXO for Alice, and possibly a fee VTXO to O.

Alice signs $\texttt{tx}_A^{\texttt{ark}}$ and asks O to co-sign. Once signed, she sends the transaction to Bob along with the VTXT path to $\texttt{vtxo}_A$, enabling Bob to exit $\texttt{vtxo}_B$ unilaterally. In a

way, Bob now holds his own VTXO, without needing prior funds in Ark or on Bitcoin.

However, there is a critical issue: Bob must trust Alice not to double-spend $\texttt{vtxo}_A$. She could unilaterally exit with it (forcing Bob to constantly monitor the chain and post $\texttt{tx}_A^{\texttt{ark}}$ if Alice ever exits with $\texttt{vtxo}_A$) or collude with O to create conflicting Ark transactions spending the same VTXO. Since Ark transactions are virtual, Alice and a (malicious) O could theoretically spend the same VTXO arbitrarily many times. If Alice broadcasts $\texttt{vtxo}_A$ onchain, only one such transaction could be confirmed. Next, we explain how Bob can secure his funds by batch swapping $\texttt{vtxo}_B$ for a new VTXO in a batch confirmed onchain.

3.3. Batch Swaps

To finalise Alice's payment, Bob *atomically* swaps $\texttt{vtxo}_B$ for a new VTXO, a process we call *batch swapping*. Here, O takes $\texttt{vtxo}_B$ and issues Bob a new VTXO $\texttt{vtxo}_{B'}$ in the next batch, which serves as onchain confirmation that Bob owns value b. This guarantees that either both transfers succeed or both fail, removing the need for Bob to monitor the chain or trust O not to collude with Alice. For clarity we describe one-to-one VTXO swaps, though the protocol supports swapping multiple VTXOs into arbitrary new allocations.

A batch swap starts with Bob requesting O to swap $\texttt{vtxo}_B$. O builds a transaction $\texttt{tx}_O$ spending its own funds and producing outputs $\texttt{tx}_O.\texttt{outs} = [(b', \texttt{chkMulSig}_{2,2;pk_O,pk_B}), (\varepsilon, \texttt{chkSig}_{pk_O})]$. This serves as a basic commitment transaction (formally defined in §3.4). The first output represents a batch; the second is an *anchor output* with dust value ε, essential for atomicity.

O also creates a virtual transaction $\texttt{tx}^{\texttt{virtual}}$ spending the first output of $\texttt{tx}_O$ and producing $\texttt{vtxo}_{B'}$. O signs $\texttt{tx}^{\texttt{virtual}}$ and shares it with Bob along with the anchor output. Bob then constructs and signs a *forfeit transaction* $\texttt{tx}^{\texttt{forfeit}}$ spending both $\texttt{vtxo}_B$ and the anchor, sending all funds to O. The signature uses the `SIGHASH_ALL` flag and is only valid for this specific transaction spending the anchor. Thus, $\texttt{tx}^{\texttt{forfeit}}$ is valid only if both $\texttt{tx}_O$ and $\texttt{vtxo}_B$ are onchain. O can now safely broadcast $\texttt{tx}_O$, as the anchor ensures atomicity. If Bob turns $\texttt{vtxo}_B$ into a UTXO, O can claim b via $\texttt{tx}^{\texttt{forfeit}}$. On the other hand, O is unable to claim $\texttt{vtxo}_B$ using $\texttt{tx}^{\texttt{forfeit}}$ as long as $\texttt{tx}_O$ is not included onchain.

Bob has successfully swapped $\texttt{vtxo}_B$ for $\texttt{vtxo}_{B'}$, which he can now unilaterally exit with by posting $\texttt{tx}^{\texttt{virtual}}$, regardless of whether Alice tries to unilaterally exit with or double-spend $\texttt{vtxo}_A$. He no longer needs to monitor the blockchain for Alice turning $\texttt{vtxo}_A$ into a UTXO. If she does, it is now O's responsibility to broadcast the Ark and forfeit transactions to claim its funds.

This construction understandably seems cumbersome. However, its power becomes clear when realising we can process more than one VTXO in this way. Indeed, the first output of $\texttt{tx}_O$ is just a batch containing one VTXO (where the VTXT consists only of $\texttt{tx}^{\texttt{virtual}}$). We can increase the size of this batch, including other VTXOs that may also have been created through batch swaps.

Finally, note that the current construction also locks O's funds, since Bob's signature is required. We show next how these funds are later released to O.

3.4. Commitment transactions

The previously defined $\texttt{tx}_O$ offers a natural way to contain batches as its outputs. Moreover, the second output of $\texttt{tx}_O$ enables to swap VTXOs atomically. This second output takes on the role of a so-called *connector*.

Definition 3.5 (Connector)**.** *A* connector *is a transaction output which is locked by a Taproot script* `connectorScript` *with an unspendable key path and a script path that specifies spending according to a VTXT where the root spends the full connector, and where each leaf of the VTXT has an anchor output as its only output. The remaining nodes of the VTXT are virtual transactions with a connector as their only output. Each output can be spent by a signature from O.*

In the Ark protocol, a connector thus encapsulates all the anchor outputs serving as inputs to forfeit transactions that can only be included onchain if the commitment transaction containing that connector is included onchain. Unlike batches, the virtual transactions of a connector are signed solely by O.

A commitment transaction $\texttt{tx}^{\texttt{commit}}$ is now simply a transaction with at least one batch or one connector output. It may output multiple batches and connectors. Additionally, it may contain inputs and outputs related to users joining (*boarding*) and leaving (*exiting*) the Ark.

Boarding. Consider a user Alice with some onchain funds locked in a UTXO $\texttt{out}_A$. Alice can join, or *board* the Ark via a two-step process. First of all, Alice will construct and broadcast the *boarding transaction* $\texttt{tx}^{\texttt{board}}$ with input $\texttt{out}_A$ and output $\texttt{out}_A^\star$, where the locking script of the output $\texttt{out}_A^\star$ can be written as $\texttt{Taproot}(\texttt{False}; \texttt{chkSig}_{pk_O \oplus pk_A}, \texttt{chkSig}_{pk_A} \wedge \texttt{relTlk}(t_b))$. This is a Taproot script with an unspendable key path and two script paths. The first script path is an *exit* path, which lets Alice spend her funds after a timeout period t_b, and the second is a *cooperative* path, in which Alice and O spend the funds together.

Alice can now send a boarding request to O, who will first verify that $\texttt{out}_A^\star$ cannot be spent by Alice alone (preventing a double-spend of the next commitment transaction $\texttt{tx}^{\texttt{commit}}$), and then create one or more VTXOs for Alice in a batch of the $\texttt{tx}^{\texttt{commit}}$. In return, $\texttt{out}_A^\star$ is added as an input to $\texttt{tx}^{\texttt{commit}}$ via the cooperative path of $\texttt{out}_A^\star$. In case Alice would not want to board the Ark after all, she can either spend $\texttt{out}_A^\star$ via the exit path after the timeout, or cooperate with O to spend $\texttt{out}_A^\star$ for a new UTXO that can be spent by Alice only.

Leaving. Alice holds one or more VTXOs in confirmed batches and can exit either *unilaterally* or *collaboratively*. In

the unilateral case, she simply broadcasts $\texttt{path}(\texttt{vtxo})$ for each owned $\texttt{vtxo}$, without interacting with O. In the collaborative case, Alice batch swaps her VTXOs, but instead of receiving new ones, O includes a UTXO for her in the next commitment. As in §3.3, O also obtains a signed forfeit transaction to prevent Alice from reusing old VTXOs.

Constructing a fully signed commitment requires user coordination (see Figure A.1 and Appendix E). This process assumes responsive participants[2]. Figure 2 illustrates: P_1 boards via a boarding transaction; O combines P_1's and its own funds into a commitment; P_4 swaps a VTXO; and P_5 exits. The batch then includes a VTXO for P_1, a new VTXO for P_4, a UTXO for P_5, and a connector allowing O to reclaim P_4's and P_5's old VTXOs if posted onchain.

3.5. Ark transactions revisited

The earlier testnet version of the Ark protocol is fully described by §3.1-§3.4. However, we found out two critical vulnerabilities in this protocol, which we call the *hostage attack* and *spam attack*.

Hostage attack. Recall from §3.1 that O is responsible for broadcasting Ark and forfeit transactions when already spent VTXOs appear onchain. A problem arises when an Ark transaction includes inputs from multiple batches: at some point, some batches will have expired whereas others have not. O wants to sweep expired batches, but doing so exposes it to users who try to unilaterally claim unexpired VTXOs. Normally, O would broadcast the Ark transaction in response, then the forfeit transaction to claim batch swapped output VTXOs. However, once a batch containing an input VTXO has been swept, the Ark transaction becomes invalid and cannot be included onchain. O has thus no way to claim this spent VTXO, resulting in a loss of funds. A malicious user could hold O's funds hostage for prolonged periods of time by having Ark transactions with inputs from many different batches. For example, if a user has an Ark transaction spending from batches expiring at T_1 and $T_2 > T_1$, and an Ark transaction spending from batches expiring at T_2 and $T_3 > T_2$, then O cannot sweep any batch until T_3. This attack can be prolonged as long as the operator processes Ark transactions spending from multiple batches. If the operator would not do so this would severely limit the Ark's functionality.

Spam attack. Recall from §3.1 that a forfeit transaction is only signed for a VTXO that is batch swapped. Hence, a malicious user Mallory holding a VTXO $\texttt{vtxo}_M$ could, in theory, make a chain of Ark transactions to himself, each time spending the new VTXO in the next transaction, and batch swap or cooperatively exit the final VTXO $\texttt{vtxo}_{M'}$. When the new commitment is confirmed, he could then exit $\texttt{vtxo}_M$ unilaterally. The operator now has to post all Ark transactions onchain, incurring onchain fees to finally broadcast the forfeit transaction that spends $\texttt{vtxo}_{M'}$. As the potential Ark transaction fees will be significantly lower than the corresponding onchain fees, this spam attack could make it unprofitable for O to broadcast the transaction chain up to the forfeit transaction, allowing Mallory to illegitimately claim both $\texttt{vtxo}_M$ and $\texttt{vtxo}_{M'}$, essentially stealing $\texttt{vtxo}_{M'}.\texttt{value}$ from the operator.

Reset transactions. To defend against these attacks, we introduce an additional virtual transaction, the *reset transaction*, which allows O to claim spent VTXOs even if the corresponding Ark transaction became invalid after an earlier sweep. This transaction effectively shifts the responsibility of posting the subsequent Ark transaction to users.

Consider the setting of §3.2, where Alice sends an amount p to Bob. In addition to constructing an Ark transaction as before, she also creates a reset transaction spending $\texttt{vtxo}_A$ collaboratively, producing one output with value p and locking script $\texttt{Taproot}(\texttt{False}; \texttt{chkSig}_{pk_O \oplus pk_A}, \texttt{chkSig}_{pk_O} \wedge \texttt{absTlk}(T_e)))$. Her Ark transaction then spends this output, giving p to Bob. The reset output can thus be spent either jointly by Alice and O or swept by O once the batch expires. Alice first signs the Ark transaction and sends both the Ark and reset transactions to O, who verifies the scripts, signs both, and returns them. Alice then signs the reset transaction and passes it to O[3]. The overall back-and-forth is shown in Figure 3. The resulting transaction flow is also illustrated in Figure 2, where P_3 sends P_4 the amount v_4, which P_4 then batch swaps.

If Alice wants to perform an Ark transaction with multiple inputs, she first constructs a reset transaction for each input VTXO, and builds an Ark transaction spending the outputs of all these reset transactions. This protects O for the hostage attack: if VTXOs from different batches are spent by an Ark transaction, expired batches can safely be swept. Indeed, if an input VTXO appears onchain, O can simply broadcast the corresponding reset transaction, claiming the funds at that batch's expiry. The Ark transaction, which requires all input VTXOs and their corresponding reset transactions to be onchain, became invalid once the expired batches were swept. A receiver wishing to exit unilaterally with the outputs of the Ark transaction should thus do so before any of the batches holding an input to the Ark transaction expires.

In case of a spam attack, where Mallory exited unilaterally with $\texttt{vtxo}_M$, the operator will now simply broadcast the appropriate reset transaction, forcing Mallory to post the subsequent Ark transaction. The operator will always force Mallory to post the next Ark transaction, until eventually the operator can post the forfeit transaction. Mallory having to post all the Ark transactions to try to claim $\texttt{vtxo}_M$ should deter Mallory from trying to claim both $\texttt{vtxo}_M$ and $\texttt{vtxo}_{M'}$. Note that this deters any rational attacker from mounting this attack. However, a Byzantine user could mount this attack regardless. To protect against such attackers, O could additionally limit the length of a chain of Ark transactions, forcing the newest holder to batch swap instead.

2. If a user is offline, the transaction aborts but can be retried without that request.

3. More precisely, both cooperatively produce the aggregated signatures for the Ark and reset transactions, in that order.

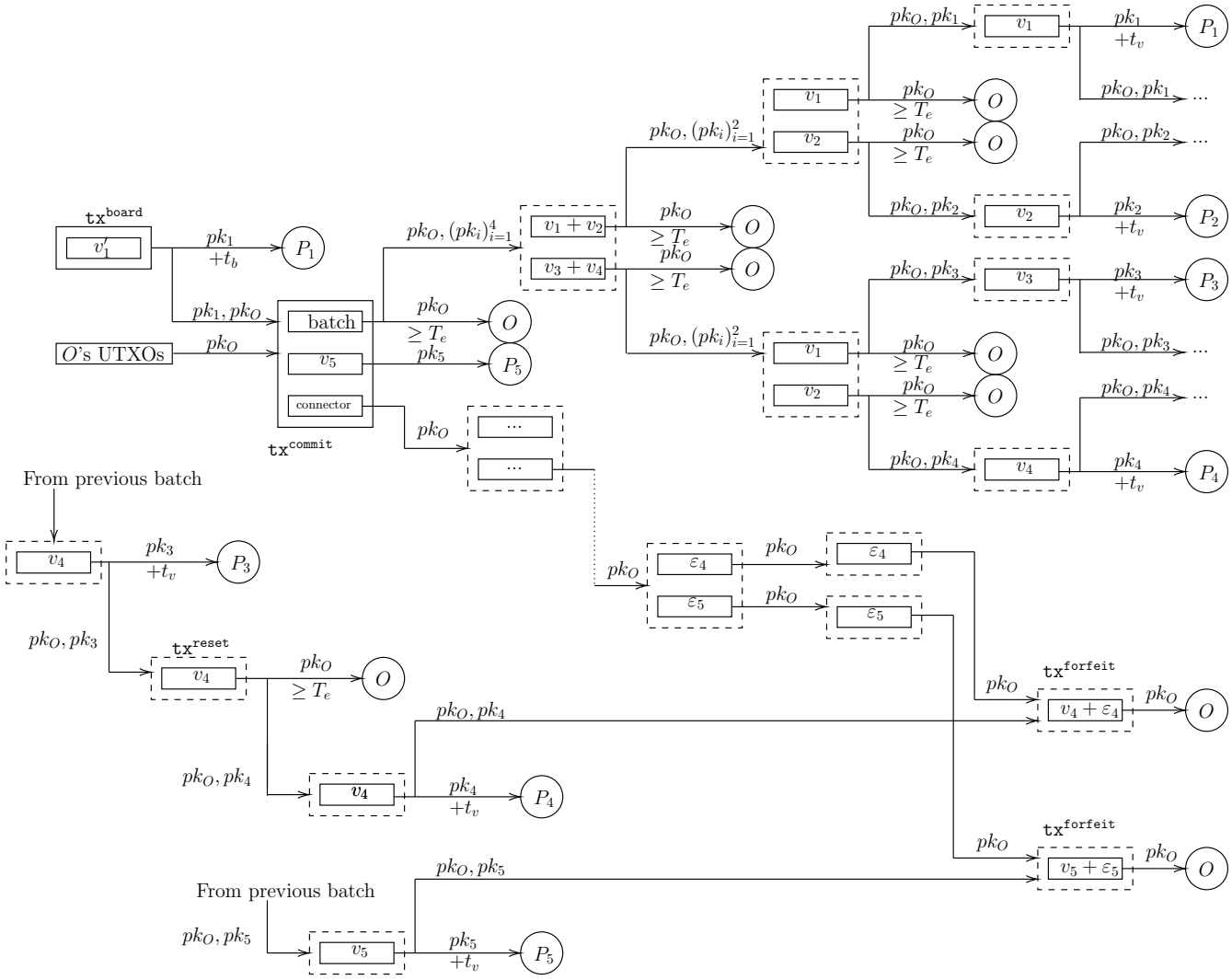

Figure 2: Transaction dependencies within the Ark protocol. Dashed transactions are virtual transactions and optimistically never appear onchain. We set $T_e = h + 2k + t_e$, where h is the block height $\texttt{tx}^{\texttt{commit}}$ got submitted by O and t_e the time in which a batch should expire. Moreover, recall that t_v is the minimum delay for a unilateral VTXO exit compared to any collaborative exit, and t_b the boarding transaction timeout period.

3.6. Offchain handover

As described in §3.4, unilateral exits involve broadcasting the whole virtual transaction path leading up to the to-be-exited VTXO. Onchain activity may thus spike if many Ark users unilaterally exit their VTXOs, for example when an operator stops responding and users must claim funds to prevent loss. Even if the operator remains active, facilitating collaborative exits can still create numerous onchain UTXOs when many users are involved. If some users wish to stay in an Ark, such exits require a subsequent onchain boarding transaction into a new Ark. To avoid this footprint, we introduce an offchain handover procedure, allowing a VTXO in an Ark with operator O_1 to move into an Ark with operator O_2. If O_1 plans to shut down but stays online to process handovers, its users can migrate to O_2's Ark entirely offchain, with the only onchain footprint being O_1 reimbursing O_2 for the transferred VTXOs.

Consider a user Alice holding a VTXO $\texttt{vtxo}_A$ of value v in a confirmed batch funded by O_1. Suppose O_1 plans to shut down its Ark but remains cooperative to transfer Alice's VTXO to an Ark run by O_2. This transfer is effectively a batch swap with connector spendable by O_1, and the new VTXO funded by O_2.

Alice requests a batch swap to O_2, as in §3.3. O_2 includes a new VTXO for Alice and an anchor output in its commitment transaction, but this anchor is now spendable by O_1. Instead of Alice and O_2 signing a forfeit transaction spending the anchor and giving $\texttt{vtxo}_A$ to O_2, Alice and O_1 sign one giving it to O_1. This two-operator batch swap ensures O_1 can claim $\texttt{vtxo}_A$ if Alice exits unilaterally. Since O_2 funds Alice's new VTXO without being able to

sweep $\texttt{vtxo}_A$, O_1 must reimburse O_2 by amount v. This is conditional on O_2's commitment transaction being confirmed onchain. To enforce this atomicity, O_2's commitment includes an anchor output $\texttt{out}_{1\to2}$ used as input to $\texttt{tx}_{1\to2}$, where O_1 pays O_2 the amount v. Before signing this modified commitment transaction, O_2 requests $\texttt{tx}_{1\to2}$, signed by O_1. If valid, O_2 can broadcast both transactions and safely claim v. Conversely, O_1 only pays if O_2's commitment is confirmed onchain. This scheme scales to multiple VTXOs, where $\texttt{tx}_{1\to2}$ now pays the total amount O_2 needed to fund all the transferred VTXOs. For the users, the online requirement is the same as for a normal batch swap (except that some tasks are in collaboration with O_1, and others with O_2). Additionally, we need O_1 to remain responsive and handle the extra overhead of signing off on $\texttt{tx}_{1\to2}$ and verifying O_2's commitment.

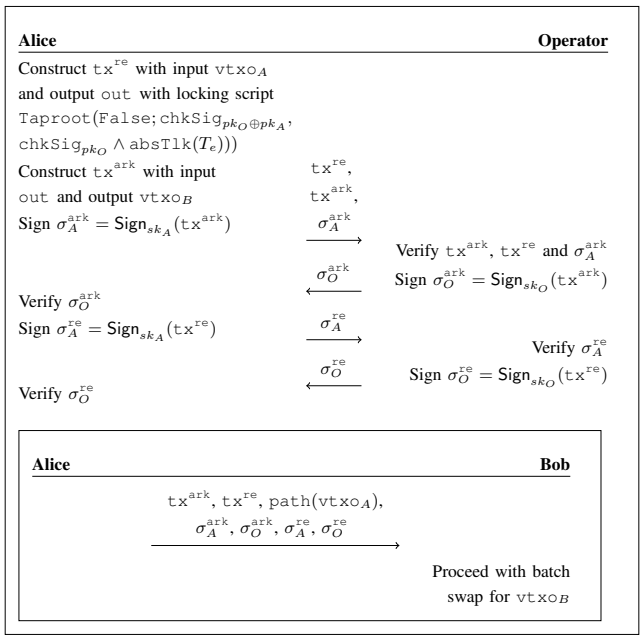

Figure 3: Alice sends funds to Bob via an Ark transaction. Upon receiving the signed Ark transaction $\texttt{tx}^{\texttt{ark}}$ and reset transaction $\texttt{tx}^{\texttt{re}}$ (described in §3.2 and §3.5 respectively), Bob will request a batch swap as introduced in §3.3 and only consider the transaction finalised once the corresponding commitment transaction is confirmed onchain.

4. Fast Finality

In §3.3 Bob must batch swap the VTXO received from Alice before treating the payment as final: ownership is only guaranteed once the corresponding commitment confirms onchain, since Alice could otherwise collude with O. This limits how quickly VTXOs can be spent again.

Assuming an *honest* operator, Bob need not fear collusion: an honest O will not co-sign a second spend of an already spent VTXO, so users can accept and re-spend VTXOs immediately, without waiting for the next commitment. To guarantee honest operator behaviour, we aim at deterring the operator from misbehaving using financial punishments. That is, we seek to realise *fast finality* under the following assumptions:

(A1) O is *rational*, i.e., maximising profits,
(A2) there is a publicly known committee of n signers, with at least one honest signer[4],
(A3) Ark users monitor the chain as long as they hold VTXOs not committed to in a confirmed batch,
(A4) at all times, there is at least one honest user online.

In the remainder of this section, we present a design incentivising O to behave honestly. This design aims at burning funds as an economic punishment for misbehaviour, in particular, for signing off on more than one Ark transaction spending the same VTXO. By (A1), O will not misbehave as long as doing so is not profitable due to this punishment. On a high level, O puts up an onchain collateral of size c (which we quantify later), that can be burned in case of misbehaviour.

Assuming O is now incentivised to behave honestly, there is one risk remaining: that a previous owner exits unilaterally, trying to claim funds it already spent offchain through Ark transactions. This is however solved by (A3). Every unilateral path is gated by a relative timelock, allowing the current holder to immediately post the collaborative Ark transactions that spend the contested VTXO (bringing other inputs onchain as needed, which one should note may be costly) and thus win the race. By (A3), each receiver monitors the chain until they spend again, ensuring a prior owner will not be able to claim the funds of the current owner. Note that when the receiver spends again, the new owner (in the case of a batch swap or cooperative exit, O) assumes monitoring responsibility.

We therefore from now on focus on incentivising honest operator behaviour. To do so, we first discuss the general misbehaviour punishment mechanism. Then, we describe how misbehaviour can be detected under our assumptions. This will ease the presentation of our design afterwards.

4.1. Punishing double-signing: Schnorr nonce reuse

O is deterred from double-signing by having a sufficient amount of its funds burned. On a high level, this mechanism works as follows. First, operator funds have to be locked in a UTXO that can either be spent by O alone after a timelock t_p, or immediately by a (burn) transaction signed by O and co-signed by an n-of-n committee that *burns* the funds (e.g., via `OP_RETURN`). (A2) guarantees that at least one of the signers in the committee behaves honestly, which in this case means that this signer deletes its keys after signing, and makes sure the burn transaction containing all committee signatures is publicly known. This guarantees that before t_p, the operator funds can only be burned. Moreover, this can only be done by someone knowing the operator's private

4. We assume that this committee is fixed throughout the Ark's lifetime and its identity is known to all Ark users. We leave for future work how to assemble this committee, and how to implement other designs such as rotating committees.

key. Our aim is now to devise a mechanism that reveals that private key in case of operator misbehaviour.

A first idea is to use accountable assertions [39] or DAPS-based schemes [40] to enable such private key extraction if O double-signs. However, these schemes are currently not implementable in Bitcoin. Instead, we take advantage of Schnorr signature nonce reuse: given two valid Schnorr signatures $\sigma_{1,2} = (R_{1,2}, s_{1,2})$ on distinct messages $m_1 \neq m_2$ under the same key pk, sk can be recovered if $R_1 = R_2$ (see Appendix B for details). We construct VTXOs so that their *collaborative* path requires O's signature with a *fixed* nonce commitment R^* embedded in the script. Thus, if O ever co-signs two different transactions spending the same VTXO, the reused nonce reveals sk, enabling anyone to burn the operator's collateral. Concretely, the collaborative script additionally checks the provided Schnorr signature under public key pk is valid and uses the committed nonce R^*[5]:

$$\texttt{OP\_DUP OP\_DUP } pk \texttt{ OP\_CHECKSIGVERIFY}$$
$$R^* \| 00^{32} \texttt{ OP\_GREATERTHANEQUAL OP\_SWAP}$$
$$R^* \| \texttt{FF}^{32} \texttt{ OP\_LESSTHANEQUAL OP\_AND}.$$

An honest receiver accepts a fast finality payment only if the received Ark transaction and all its predecessors spend VTXOs whose collaborative paths commit to specific nonces. If O were to later double-sign on any of those VTXOs, we want to guarantee that this misbehaviour is detected by an honest user. This would allow the honest user to extract the operator key and sign off on the corresponding burn transaction, punishing O.

4.2. Detecting double-signing: the Ark ledger

A double-sign, i.e., two fully signed Ark transactions sharing (at least) one VTXO input, would be apparent if honest users can access a ledger of all Ark transactions made so far. We call such a ledger an *Ark ledger*, and it can simply be seen as a list of fully valid Ark transactions.

Definition 4.1 (Ark ledger)**.** *An* Ark ledger $\mathcal{A}$ *is a set of Ark transactions that (i) carry valid witnesses for all their VTXO inputs, (ii) have as inputs VTXOs created by other transactions in $\mathcal{A}$ or VTXOs committed to in a batch output confirmed onchain that has not yet been swept.*

Instead of having this Ark ledger stored in one central (trusted) location, we require each Ark user to maintain its own Ark ledger. For now, we assume a fixed list $N^{\texttt{ff}}$ of Ark users that transact in Ark with fast finality. Since there is no guarantee that *every* Ark user will share its Ark transactions, we argue how, by (A3), each honest Ark user maintaining an Ark ledger containing all transaction paths shared by users in $N^{\texttt{ff}}$, detects any double-sign defrauding an honest user in $N^{\texttt{ff}}$. We deem this a sufficient security property, as we do not care about malicious users defrauding each other. Indeed, we only want to prevent two honest sellers providing a service or goods to a double-spending coalition of buyer and operator.

To see this, we distinguish three cases, each concerning two Ark users P_1, P_2 receiving two VTXOs $\texttt{vtxo}_1$, $\texttt{vtxo}_2$, respectively, where $\texttt{path}(\texttt{vtxo}_1)$ and $\texttt{path}(\texttt{vtxo}_2)$ contain transactions $\texttt{tx}_1^*$ and $\texttt{tx}_2^*$, respectively, which both spend the same VTXO.

1) P_1, P_2 are both honest.
2) P_1 is honest, P_2 is malicious, without loss of generality.
3) P_1, P_2 are both malicious.

In case 1, as honest users broadcast their Ark transaction paths, both P_1 and P_2 detect the operator misbehaviour when they learn each other's conflicting transaction paths.

In case 2, we only know for sure that P_1 broadcast its transaction path. If P_2 does so too, we are in case 1. Otherwise, we again have different cases: both P_1 and P_2 can try to batch swap, exit (unilaterally or collaboratively), or spend (in an Ark transaction) their respective VTXOs. Let us enumerate all possible actions P_1 can take, and see how P_2 may act for each situation.

(a) P_1 *sends a batch swap request.* O can incorporate the request in a commitment transaction and give P_1 a fresh VTXO. O funded this fresh VTXO itself, and P_1 now enjoys the security guarantees of a VTXO that is part of an onchain, confirmed batch. However, a rational operator would not accept P_2 batch swapping or exiting $\texttt{vtxo}_2$, as O would have to fund two new VTXOs and only be able to claim one old VTXO. If P_2 exits unilaterally, P_1 will observe a conflicting transaction onchain, detecting misbehaviour. If P_2 just holds the VTXO until expiry, the VTXO will expire and be swept, as if the double-sign never occurred. If P_2 instead spends the funds via an Ark transaction, we assume without loss of generality that the receiver is honest (as otherwise we end up in either one of the previous cases where the malicious holder tries to batch swap or exit, or just holds the VTXO until expiry). But then, we end up in case 1, for which we know detection will occur.

(b) P_1 *requests a cooperative exit.* Same analysis as case (a).

(c) P_1 *performs an Ark transaction.* P_1 is no longer the one being defrauded. Instead, the new owner of the VTXO funds is. We are only interested in the case where the new owner (or another future owner further down the chain of possession) is honest, as this honest user is now at risk of being defrauded. We are then once again in case 2. However, batch expiry got closer, meaning that we do not need to analyse the case where the honest user performs Ark transactions indefinitely. Eventually, batch expiry will get close enough and an honest user will no longer consider an Ark transaction, but either a batch swap or an exit. This brings the analysis in one of the remaining cases.

(d) P_1 *exits unilaterally.* Once again, a rational operator will not accept a batch swap or collaborative exit request from P_2. If P_2 also exits unilaterally, by (A3) P_1 monitors the chain the conflicting transaction will become

5. For now we would have to verify the user's signature separately. Combining nonce commitments with key aggregation is left for future work.

known to P_1, meaning the misbehaviour is detected. Finally, if P_2 performs an Ark transaction with the funds, we can again argue that only the case of an honest receiver is of interest, resulting in immediate detection as well.

Finally, case 3 is only of interest if at some point at least part of the funds locked in $\texttt{vtxo}_1$ or $\texttt{vtxo}_2$ VTXO is spent to an honest user. If this happens, case 3 reduces to either case 1 or case 2. Otherwise, no honest user is being defrauded, in which case we are not interested in detecting the misbehaviour. Again, such a case would involve a deviating operator with one or more malicious users defrauding each other, which does not affect honest users in any way.

We have thus argued how, for a fixed set $N^{\texttt{ff}}$, honest users in $N^{\texttt{ff}}$ maintaining an Ark ledger can always detect misbehaviour which would defraud an honest user in $N^{\texttt{ff}}$, assuming (A3). Our design allows honest users to safely accept fast finality payments, as long as they are in the set $N^{\texttt{ff}}$. In this work, we give O the responsibility of periodically publishing $N^{\texttt{ff}}$, specifying how to reach each Ark user accepting fast finality payments. $N^{\texttt{ff}}$ could be put onchain, or on another publicly accessible location (with for example an onchain commitment to $N^{\texttt{ff}}$). Note that the only power this responsibility comes with is determining which honest users can safely accept fast finality payments. An honest user will itself check whether it is in the current $N^{\texttt{ff}}$. If so, it will accept fast finality payments and protect itself against double-spends by broadcasting any transaction path it receives to the other users in $N^{\texttt{ff}}$.

4.3. Incentivising honest behaviour

Our design assumes the operator locks an amount c into a UTXO which can either be spent again by O after a timelock, or by burning it before that timelock. This requires the cooperation of the 1-of-n honest signer committee exactly once, to set up this single burn transaction. Afterwards, the operator is incentivised to behave honestly as long as the cheating gains are smaller than c. Suppose the honest Ark users are maintaining an Ark ledger $\mathcal{A}$, and no misbehaviour has yet been detected. In the worst case, there is for every Ark transaction in $\mathcal{A}$ another transaction spending the same funds, signed off on by O. Denote by v_{spent} the total amount of funds spent by all Ark transactions in $\mathcal{A}$, i.e., $v_{spent} = \sum_{\texttt{tx} \in \mathcal{A}} \texttt{tx.ins.value}$. Then v_{spent} is an upper bound on the profit that could be made by defrauding the honest users. As long as $v_{spent} < c$, a rational operator would not engage in double-signing, as it knows by (A4) misbehaviour will be detected and punished, burning c.

Hence, as long as c is chosen large enough in order to exceed v_{spent}, which can be interpreted as the transaction volume within the Ark, honest users can be assured no double-signing will occur. If v_{spent} ever grows within some margin of c, it is up to the honest users to stop transacting via Ark transactions, instead cementing their VTXOs through batch swaps before proceeding, or waiting for O to put up more collateral. Future research should determine what this margin must realistically be, as a function of the network latency Δ and the transacting frequency within the Ark.

Remark 4.2. *Notice that that v_{spent} shrinks whenever a batch gets swept. In other words, the collateral requirement grows larger with larger batch expiry times. This indicates the important trade-off between collateral and the user online requirement, as a lower collateral requires more frequent batch expiries, and thus more frequent batch swapping by users to preserve funds. It will be up to the free market to find a balance between collateral and batch expiry.*

In Appendix C, we outline an alternative design that does not require any collateral from the operator. The trade-off is that the 1-of-n honest signer committee has to be involved for every Ark transaction, instead of only when setting up the burn transaction for the collateral. The signer committee can in turn be eliminated in case Bitcoin would add covenants to its scripting capabilities.

5. Security Analysis

This section argues informally why Ark satisfies the properties of §2.3. Formal statements and proofs of the security properties appear in Appendix F.

VTXO Security, Theorem F.7. (i) *Safety.* Any $\texttt{vtxo}$ in a confirmed, unexpired batch cannot be spent without a valid witness, as long as one cosigner on $\texttt{path}(\texttt{vtxo})$ is honest. An honest cosigner never signs conflicting transactions, emulating a covenant: before expiry, a VTXO is either exited unilaterally or spent collaboratively with O, in which case it is only considered spent when there is a valid Ark or forfeit transaction, requiring a valid witness. (ii) *Liveness.* For any $\texttt{vtxo}$ in a confirmed batch, if at least one cosigner on $\texttt{path}(\texttt{vtxo})$ is honest so that $\texttt{vtxo}$ remains unspent, then anyone knowing $\texttt{path}(\texttt{vtxo})$ can broadcast it to exit unilaterally. This is safe up to shortly before batch expiry, leaving $2k$ blocks (blockchain depth parameter) to ensure all transactions in the path confirm onchain.

Ark Atomicity, Theorem F.12. For any action involving an honest party (user or operator), either the state changes exactly as intended or not at all. We show this by going through each honest Ark action and making sure honest participants can abort at any time upon detecting misbehaviour and only provide witnesses for transactions they approve, ruling out partial execution.

Balance Security, Theorems F.10 and F.13. (For a user) A user's balance consists of funds in boarding outputs and VTXOs that can be claimed without O's help (formally defined in Appendix F). Applying VTXO Security ensures that an honest user, holding the necessary paths, can always reclaim these funds via unilateral exit. (For an operator) O funds each commitment but eventually reclaims its inputs once the commitment expires. By atomicity, every coin spent by an honest operator corresponds to at least as much recoverable value in the sweep. Tracking all flows shows that after a phase of serving exits, all operator funds return.

Scalability. (i) *Constant Updates.* A batch can in principle commit to an arbitrarily large VTXT. Only the root of the

tree appears onchain as part of the commitment transaction. The connector output can also contain arbitrarily many connectors. Note, however, that to confirm each successive VTXO spend, the new VTXO must appear in its own batch, requiring an additional onchain commitment. (ii) *Constant optimistic exit.* With a cooperating operator, a user can exit multiple VTXOs collaboratively as an additional output in a commitment transaction. (iii) *Logarithmic pessimistic exit.* A unilateral exit requires posting `path(vtxo)`, which has length $\mathcal{O}(\log n)$ in a VTXT with n VTXO as its leaves.

6. Implementation and Evaluation

The open-source mainnet implementation of the Ark protocol[6] has incorporated our reset transaction scheme to defend against the hostage and spam attacks described in §3.5. We performed a number of experiments with this Ark implementation to quantify on one hand the time required to construct a commitment transaction and on the other hand the onchain footprint of commitment transactions and unilateral exits.

First, we study the effect of batch sizes on the time it took to finalise the commitment transaction. We have performed an experiment where an Ark operator receives from each of n users exactly one batch swap request, for $n = \{2^1, \dots, 2^7, 200\}$. In Table 2, we present the times it took for every n for (BC) O to build and send out to all users a commitment transaction with one batch and one connector, (SS) all signing sessions for transactions in the batch's VTXT, and (FF) the users to construct forfeit transactions and have them fully signed with O. It then took O on average 0.023 seconds to sign the commitment transaction.

TABLE 2: Execution times t in seconds per value of n for each step (with R^2 for linear regression $t = a_0 + a_1 n$, and p-value for $H_0 : a_1 = 0$ against $H_1 : a_1 \neq 0$).

$t(n)$	2	4	8	16	32	64	128	200	R^2	p
BC	0.030	0.049	0.053	0.080	0.084	0.220	0.427	0.806	0.981	2.25e-6
SS	0.190	0.194	0.200	0.219	0.333	0.388	0.707	1.075	0.991	2.01e-7
FF	0.114	0.123	0.132	0.169	0.224	0.370	0.556	0.831	0.998	4.71e-9

The time each step takes seems to grow linearly with n (Musig2 is $\mathcal{O}(n)$ [38]). Looking at $n = 200$, all users must be online *simultaneously* for about 2.7 seconds. In practice, users request service over time and may go offline. O collects these requests and eventually starts building a commitment transaction, prompting involved users to come back online. The advantage of Ark is that updates require only the relevant users, not everyone, enabling O to use smaller, more frequent batches with reduced interactivity. Note that this experiment works with minimal network delay; real-world communication would be slower.

The experiment returns the commitment transaction and the batch's VTXT. For any node in the VTXT, one can retrieve the corresponding PSBT data and decode it to JSON (see Appendix D). For any `vtxo`, we examine `path(vtxo)` and observe consistent transaction sizes: 107 vB for the leaf (`P2TR` `vtxo` and anchor), and 150 vB for each subsequent virtual transaction (2 `P2TR` outputs + 1 anchor). With fast finality, each subsequent Ark transaction would incur a similar cost. The final commitment transaction (change, batch, and connector, all `P2TR`) is 197 vB. At a fee rate of 6 sat/vB ($1\text{฿} = 10^8$ sat), unilaterally exiting `vtxo` costs 6sat/vB $\cdot$ $(\lceil \log n \rceil \cdot 150 + 107)$ sat, which for $n = 128$ would be 6942 sat.

We compare these results to other Bitcoin offchain protocols. For commitment transactions, only Coinpool and Bitcoin Clique have comparable constructions: standard Bitcoin transactions with a constant number of outputs. In terms of unilateral exit costs, a payment pool incurs constant overhead: the user spends a shared UTXO and creates two outputs (for himself and the rest). Bitcoin Clique's exit cost also grows logarithmically with the size of the clique (analogous to Ark's batch size). Finally, we found estimates for unilateral exit costs for Lightning [41] and Spark [25]. For Lightning, the expected exit cost is 181 vB $+ h \cdot 43$ vB, where h is the number of in-flight HTLCs. Spark has an exit cost of $(d + 3) \cdot 315$ vB, where d is the depth of the user's UTXO: the analog to the number of Ark transactions along the path to a VTXO, though Ark transactions may be smaller. Additionally, Ark supports batch swaps to "reset" the path length to only the virtual transactions spending the batch output.

7. Discussion and Limitations

Centralisation of Ark Operator. The Ark protocol depends on a single operator to coordinate offchain transactions, manage liquidity, and produce commitment transactions. This introduces a centralisation risk at odds with Bitcoin's decentralised ethos. Although users can unilaterally exit and retrieve funds, the operator remains a single point of failure. If unavailable or malicious (e.g., censoring users), performance degrades and onchain costs from unilateral exits rise. Future work could explore multi-operator designs via federation or cryptographic coordination, and examine how the operator's roles (funding, signing, batching) might be distributed across multiple parties. Of particular interest might be the use of (multiple) trusted execution environments (TEEs) to guarantee decentralised and honest operators.

Miner Extractable Value (MEV). The Ark operator controls which user requests are processed and how VTXOs are grouped into batches, granting sequencing power akin to a rollup sequencer. This naturally raises the question of whether the Ark operator can extract value (MEV) [42]. However, Ark's design does not introduce new scripting capabilities; thereby the operator cannot reorder or insert transactions with programmable side effects. The operator's influence is limited to scheduling (e.g., delaying or censoring transactions) as already possible in Bitcoin. Moreover, unilateral exit guarantees ensure users can reclaim funds without operator cooperation, mitigating potential abuse.

6. https://github.com/ark4fish/ARK

While this design does not have the expressive power that enables typical MEV strategies in EVM-based rollups (e.g., frontrunning, sandwiching, arbitrage), one may envision richer application logic being built on top of Ark, particularly via BitVM [30], [31]. In such cases, MEV may emerge within the application, but remains isolated. This is due to the architectural separation between the core batching mechanism and the logic enforced by BitVM: Ark merely facilitates the offchain transfer of VTXOs, while any higher-level semantics (e.g., order matching or contract execution) are enforced offchain and verified via user-side fraud proofs. Importantly, BitVM does not endow the Ark operator with additional expressive control over VTXO state or execution order. Any misbehaviour in this context is confined to participants in the application and is mitigated by standard BitVM dispute mechanisms or Ark's unilateral exit. As such, MEV originating from BitVM-based applications remains sandboxed at the application level.

Liquidity Requirements. The operator must commit upfront capital to maintain Ark's liquidity. This capital (and potential gains in operator fees) can be retrieved at the corresponding batch's expiry. Further work needs to quantify liquidity needs under real-life market conditions, to develop trustless or trust-minimised ways for external liquidity providers to fund commitment transactions, and to design incentives such as fee markets to attract said liquidity providers. Additionally, a fast finality mechanism requires an operator collateral that exceeds any gain that could be made from double-signing Ark transactions. This gain scales with the coins that can be double-spent before detection. Further research should look into alternative collateral constructions, and quantify and integrate other incentives, such as the loss of future revenue that would deter operators from cheating.

Onchain cost of unilateral exit. In a unilateral exit, users must broadcast virtual transactions to claim their VTXO onchain and pay associated fees. Fast finality increases this cost, as all prior Ark transactions leading to the exited VTXO must also be put onchain. Future work may explore VTXO designs that reduce exit costs for small holders and evaluate policies limiting successive Ark spends before a batch swap, and their impact on throughput. Compared to rollups, which always require all transactions onchain, this unilateral exit cost is an improvement. However, protocols like Lightning have a much lower exit cost per channel, growing only with the number of in-flight HTLCs. On the other hand, the Ark operator is designed to stay online, akin to a payment channel hub. Unilateral exits are thus less likely than in Lightning, where counterparties may go offline for many reasons.

Bank Run Scenario. As for other Layer-2 protocols, a worst-case scenario for Ark is a bank run, where many users exit simultaneously. This triggers a surge in transactions proportional to the number of VTXOs in affected batches, potentially overwhelming Bitcoin's limited throughput and delaying exits. Users should therefore initiate unilateral exits well before batch expiry to avoid race conditions with operator sweeps. Future research should examine how quickly a bank run can be processed onchain based on the number of exited VTXOs. Finally, it is worth recalling §3.6, which showed that in case an operator wants to halt operations but remains responsive, users of that Ark can migrate their VTXOs to a new Ark in an offchain manner. An Ark can thus be wound down in a controlled manner, even avoiding collaborative exits of every Ark user (which has an onchain footprint).

8. Conclusion

Ark is the first Bitcoin-compatible commit-chain, enabling offchain transactions via an untrusted operator who batches them into succinct onchain commitments. At the core of Ark is the virtual UTXO (VTXO), a novel abstraction allowing offchain transfers with unilateral onchain settlement guarantees. Ark enables receivers of funds to onboard entirely offchain, and can update its state by coordinating only with the involved users, instead of all protocol participants.

We formally define the protocol, prove its security for both the users and the operator, and show how Ark can achieve fast finality. Our analysis led to the discovery of two vulnerabilities in the Ark protocol, which have been addressed in open-source Ark mainnet implementation. Finally, our experiments on this implementation showcase processing times linear in the number of VTXOs processed, updating the Ark state onchain with a constant 3-output transaction of 197 vB, constant cost cooperative exits (one extra output in the commitment per user) and unilateral exit costs scaling logarithmically with the number of batched VTXOs (e.g., 7 transactions of 150 vB plus a constant 107 vB for $2^7 = 128$ VTXOs).

Apart from looking into ways to decentralise the operator and reduce liquidity requirements, future work may involve analysing incentive compatibility with the base chain, exploring interoperability with solutions like Lightning, exploring which locking scripts can safely lock a VTXO, enabling otherwise impractical offchain protocols, and investigating on one hand how Ark may improve onchain privacy, and on the other hand how privacy within the Ark can be improved.

9. Ethical Considerations

We briefly outline the ethical considerations of our work. Our contribution focuses on the theoretical design and analysis of the Ark protocol, and our discussion is therefore limited to these conceptual aspects. Any real-world deployment would require implementers to evaluate additional operational, legal, and security considerations beyond the scope of this paper.

The primary stakeholders of this research include users and operators of blockchain-based systems, infrastructure developers, and the broader public that may rely on systems integrating our protocol. By improving the trust and security assumptions relative to existing approaches, our work has

the potential to reduce the likelihood of certain failure modes.

Nevertheless, risks remain. If the assumptions underlying our protocol do not hold, users may still face financial loss or exposure to unintended system behaviour. Moreover, lower trust requirements could unintentionally attract inexperienced users to complex systems whose risks they may not fully understand. While these concerns are mitigated in part by the autonomy of individuals to make informed decisions, they highlight the need for careful communication by any entity deploying an Ark.

Our research does not involve human subjects and does not constitute a production-ready system. We therefore emphasise that operators integrating our protocol into end-user products should take responsibility for providing appropriate disclosures, user protections, and harm-reduction measures. Finally, in support of transparency and reproducibility, we make all theoretical results and accompanying artifacts publicly available.

References

[1] S. Nakamoto, "Bitcoin: A peer-to-peer electronic cash system," 2008.

[2] J. Poon and T. Dryja, "The bitcoin lightning network: Scalable off-chain instant payments," 2016.

[3] C. Decker and R. Wattenhofer, "A fast and scalable payment network with bitcoin duplex micropayment channels," in *Stabilization, Safety, and Security of Distributed Systems: 17th International Symposium, SSS 2015, Edmonton, AB, Canada, August 18-21, 2015, Proceedings 17*. Springer, 2015, pp. 3–18.

[4] C. Decker, R. Russell, and O. Osuntokun, "eltoo: A simple layer2 protocol for bitcoin," *White paper: https://blockstream.com/eltoo.pdf*, 2018.

[5] G. Avarikioti, E. K. Kogias, R. Wattenhofer, and D. Zindros, "Brick: Asynchronous payment channels," *arXiv preprint arXiv:1905.11360*, 2019.

[6] M. Hearn and J. Spillman, "Contract," https://en.bitcoin.it/wiki/Contract, 2023, accessed: 2025-09-20.

[7] Z. Avarikioti, O. S. Thyfronitis Litos, and R. Wattenhofer, "Cerberus channels: Incentivizing watchtowers for bitcoin," in *Financial Cryptography and Data Security: 24th International Conference, FC 2020 , Kota Kinabalu, Malaysia, February 10–14, 2020 Revised Selected Papers*. Berlin, Heidelberg: Springer-Verlag, 2020, p. 346–366. [Online]. Available: https://doi.org/10.1007/978-3-030-51280-4_19

[8] L. Aumayr, Z. Avarikioti, M. Maffei, and S. Mazumdar, "Securing lightning channels against rational miners," in *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '24. New York, NY, USA: Association for Computing Machinery, 2024, p. 393–407. [Online]. Available: https://doi.org/10.1145/3658644.3670373

[9] E. Heilman, L. Alshenibr, F. Baldimtsi, A. Scafuro, and S. Goldberg, "Tumblebit: An untrusted bitcoin-compatible anonymous payment hub," in *Network and distributed system security symposium*, 2017.

[10] S. Dziembowski, L. Eckey, S. Faust, and D. Malinowski, "Perun: Virtual payment hubs over cryptocurrencies," in *2019 IEEE symposium on security and privacy (SP)*. IEEE, 2019, pp. 106–123.

[11] E. Tairi, P. Moreno-Sanchez, and M. Maffei, "A 2 l: Anonymous atomic locks for scalability in payment channel hubs," in *2021 IEEE symposium on security and privacy (SP)*. IEEE, 2021, pp. 1834–1851.

[12] X. Qin, S. Pan, A. Mirzaei, Z. Sui, O. Ersoy, A. Sakzad, M. F. Esgin, J. K. Liu, J. Yu, and T. H. Yuen, "Blindhub: Bitcoin-compatible privacy-preserving payment channel hubs supporting variable amounts," in *2023 IEEE symposium on security and privacy (SP)*. IEEE, 2023, pp. 2462–2480.

[13] S. Dziembowski, L. Eckey, S. Faust, J. Hesse, and K. Hostáková, "Multi-party virtual state channels," in *Annual international conference on the theory and applications of cryptographic techniques*. Springer, 2019, pp. 625–656.

[14] L. Aumayr, M. Maffei, O. Ersoy, A. Erwig, S. Faust, S. Riahi, K. Hostáková, and P. Moreno-Sanchez, "Bitcoin-compatible virtual channels," in *2021 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2021, pp. 901–918.

[15] Z. Avarikioti, Y. Wang, and Y. Wang, "Thunderdome: Timelock-free rationally-secure virtual channels," *arXiv preprint arXiv:2501.14418*, 2025.

[16] C. Burchert, C. Decker, and R. Wattenhofer, "Scalable funding of bitcoin micropayment channel networks," *Royal Society open science*, vol. 5, no. 8, p. 180089, 2018.

[17] A. R. Pedrosa, M. Potop-Butucaru, and S. Tucci-Piergiovanni, "Scalable lightning factories for bitcoin," in *Proceedings of the 34th ACM/SIGAPP Symposium on Applied Computing*, 2019, pp. 302–309.

[18] D. Chaum, "Blind signatures for untraceable payments," in *Advances in Cryptology: Proceedings of Crypto 82*. Springer, 1983, pp. 199–203.

[19] Cashu, "Cashu is Ecash for Bitcoin," 2025. [Online]. Available: https://docs.cashu.space/

[20] Fedimint, "Fedimint," 2025. [Online]. Available: https://fedimint.org/docs/intro

[21] J. Coleman, "State Channels," 2015. [Online]. Available: https://www.jeffcoleman.ca/state-channels/

[22] A. Miller, I. Bentov, S. Bakshi, R. Kumaresan, and P. McCorry, "Sprites and state channels: Payment networks that go faster than lightning," in *International conference on financial cryptography and data security*. Springer, 2019, pp. 508–526.

[23] L. Aumayr, O. Ersoy, A. Erwig, S. Faust, K. Hostáková, M. Maffei, P. Moreno-Sanchez, and S. Riahi, "Generalized channels from limited blockchain scripts and adaptor signatures," in *International Conference on the Theory and Application of Cryptology and Information Security*. Springer, 2021, pp. 635–664.

[24] R. Somsen, "Statechains: Off-chain Transfer of UTXO Ownership," 2018.

[25] Spark, "Spark documentation." [Online]. Available: https://docs.spark.money/

[26] G. Naumenko and A. Riard, "Coinpool: efficient off-chain payment pools for bitcoin." [Online]. Available: https://coinpool.dev/v0.1.pdf

[27] A. Back, M. Corallo, L. Dashjr, M. Friedenbach, G. Maxwell, A. Miller, A. Poelstra, J. Timón, and P. Wuille, "Enabling blockchain innovations with pegged sidechains," *URL: http://www.opensciencereview. com/papers/123/enablingblockchain-innovations-with-pegged-sidechains*, vol. 72, pp. 201–224, 2014.

[28] A. Kiayias and D. Zindros, "Proof-of-work sidechains," in *International Conference on Financial Cryptography and Data Security*. Springer, 2019, pp. 21–34.

[29] J. Nick, A. Poelstra, and G. Sanders, "Liquid: A bitcoin sidechain," *Liquid white paper. URL https://blockstream.com/assets/downloads/pdf/liquid-whitepaper. pdf*, 2020.

[30] L. Aumayr, Z. Avarikioti, R. Linus, M. Maffei, A. Pelosi, C. Stefo, and A. Zamyatin, "Bitvm: Quasi-turing complete computation on bitcoin," *Cryptology ePrint Archive*, 2024.

[31] R. Linus, L. Aumayr, Z. Avarikioti, M. Maffei, A. Pelosi, O. T. Litos, C. Stefo, D. Tse, and A. Zamyatin, "Bridging bitcoin to second layers via BitVM2," Cryptology ePrint Archive, Paper 2025/1158, 2025. [Online]. Available: https://eprint.iacr.org/2025/1158

[32] J. Poon and V. Buterin, "Plasma: Scalable autonomous smart contracts," *White paper*, pp. 1–47, 2017.

[33] R. Khalil, A. Zamyatin, G. Felley, P. Moreno-Sanchez, and A. Gervais, "Commit-chains: Secure, scalable off-chain payments," *Cryptology ePrint Archive*, 2018.

[34] S. Riahi and O. S. T. Litos, "Bitcoin clique: Channel-free off-chain payments using two-shot adaptor signatures," Cryptology ePrint Archive, Paper 2024/025, 2024. [Online]. Available: https://eprint.iacr.org/2024/025

[35] H. Kalodner, S. Goldfeder, X. Chen, S. M. Weinberg, and E. W. Felten, "Arbitrum: Scalable, private smart contracts," in *27th USENIX Security Symposium (USENIX Security 18)*, 2018, pp. 1353–1370.

[36] Optimism, "Optimistic Rollup Overview," 2021. [Online]. Available: https://github.com/ethereum-optimism/optimistic-specs/blob/0e9673af0f2cafd89ac7d6c0e5d8bed7c67b74ca/overview.md

[37] J. A. Garay, A. Kiayias, and N. Leonardos, "The Bitcoin Backbone Protocol: Analysis and Applications," *J. ACM*, apr 2024. [Online]. Available: https://doi.org/10.1145/3653445

[38] J. Nick, T. Ruffing, and Y. Seurin, "MuSig2: Simple two-round schnorr multi-signatures," Cryptology ePrint Archive, Paper 2020/1261, 2020. [Online]. Available: https://eprint.iacr.org/2020/1261

[39] T. Ruffing, A. Kate, and D. Schröder, "Liar, liar, coins on fire! Penalizing equivocation by loss of bitcoins," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, 2015, pp. 219–230.

[40] X. Dong, O. S. T. Litos, E. N. Tas, D. Tse, R. L. Woll, L. Yang, and M. Yu, "Remote Staking with Economic Safety," *arXiv preprint arXiv:2408.01896*, 2024.

[41] Lightning Network, "BOLT #3," https://github.com/lightning/bolts/blob/master/03-transactions.md, 2023, accessed: 2025-09-21.

[42] P. Daian, S. Goldfeder, T. Kell, Y. Li, X. Zhao, I. Bentov, L. Breidenbach, and A. Juels, "Flash boys 2.0: Frontrunning in decentralized exchanges, miner extractable value, and consensus instability," in *2020 IEEE Symposium on Security and Privacy, SP 2020, San Francisco, CA, USA, May 18-21, 2020*. IEEE, 2020, pp. 910–927. [Online]. Available: https://doi.org/10.1109/SP40000.2020.00040

[43] P. Wuille, J. Nick, and A. Towns, "Taproot: SegWit version 1 spending rules," 2020. [Online]. Available: https://github.com/bitcoin/bips/blob/master/bip-0341.mediawiki

[44] L. Aumayr, Z. Avarikioti, M. Maffei, G. Scaffino, and D. Zindros, "Blink: An optimal proof of proof-of-work," *Cryptology ePrint Archive*, 2024.

[45] Bitcoin Optech, "Cluster mempool," https://bitcoinops.org/en/topics/cluster-mempool/, 2025, accessed: 2025-09-21.

Appendix A. Global Parameters, Notation and Commitment Transaction Construction

TABLE A.1: Global parameters and Notation

P, O	Ark user and Ark operator
$\mathtt{tx}, \mathtt{out}$	Transaction, transaction output
$\mathtt{vtxo}$	Virtual transaction output
$\mathcal{L}^Q$, $\mathcal{L}^Q_{-k}$	Local view of party Q, with and without latest k blocks, denote inclusion by $\mathtt{tx} \in \mathcal{L}^P$ (or $\mathtt{out} \in \mathcal{L}^P$ for *unspent* outputs)
t_b	Boarding timeout: unilateral recovery delay for boarding funds
t_e	Batch expiry time: minimum number of blocks before batch expires
t_v	VTXO unilateral delay: minimum delay of a VTXO unilateral script path compared to any collaborative script path
ε	Dust-sized value of anchor outputs
k, u	Ledger depth/wait parameters as defined in [37]
$\mathcal{L}^Q(r)$, $\mathcal{L}^Q_{-k}(r)$	Local view of party Q at round r, with and without latest k blocks, as in [37], used only in Appendix F, denote inclusion just as before
$\mathcal{L}^1 \preceq \mathcal{L}^2$	$\mathcal{L}^1$ is a prefix of $\mathcal{L}^2$

Appendix B. Schnorr Signatures and Nonce Reuse

We show that one can extract the private key from two Schnorr signatures of different messages under the same public key, using the same nonce. To this end, let us first recall the Schnorr signature scheme, which has been introduced to Bitcoin with the Taproot upgrade.

Definition B.1 (Schnorr signatures scheme)**.** *Let λ be the security parameter. Consider the* `secp256k1` *elliptic curve group E with generator point G with order q. The Schnorr signature scheme is defined as the triple* $(\mathsf{Gen}, \mathsf{Sign}, \mathsf{Vrfy})$*, where*

- $\mathsf{Gen}(1^\lambda)$ *is the* key generation *algorithm, outputting a private-public key pair (sk, pk), where $sk \in \mathbb{Z}_q$ and $pk = skG$,*
- $\mathsf{Sign}(sk, m)$ *is the* signing *algorithm, outputting a signature $\sigma = (R, s)$ for a message m, where $R = rG$ for some randomly sampled $r \leftarrow \mathbb{Z}_q$ and $s = [r + H(R\|pk\|m)sk \bmod q]$, and*
- $\mathsf{Vrfy}(pk, m, \sigma)$ *is the* verification *algorithm, outputting 1 if σ is a valid signature of m under pk, and 0 otherwise. That is, $\mathsf{Vrfy}(pk, m, \sigma) = 1 \iff R = sG - H(R\|pk\|m)pk$.*

One can retrieve the private key sk given two valid Schnorr signatures under pk, denoted by (R_1, s_1) and (R_2, s_2), for messages m_1, m_2 ($m_1 \neq m_2$) whenever the nonce is reused, i.e., whenever $R_1 = R_2 = rG$ for some $r \leftarrow \mathbb{Z}_q$. Indeed, we have

$$\begin{aligned} &s_1 - s_2 \\ &\quad = [r + H(R\|pk\|m_1)sk - r - H(R\|pk\|m_2)sk \bmod q] \\ &\quad = [(H(R\|pk\|m_1) - H(R\|pk\|m_2))sk \bmod q]. \end{aligned}$$

From this, we can retrieve the private key

$$sk = \left[\frac{s_1 - s_2}{H(R\|pk\|m_1) - H(R\|pk\|m_2)} \bmod q\right].$$

We assume the denominator to be nonzero, as otherwise we would have found a collision of the hash function.

Appendix C. Alternative Design for Fast Finality

The lump-sum collateral approach described in §4.3 would incentivise honest users to halt transacting with fast finality in the Ark as soon as the transaction volume grew within some margin of the collateral. An alternative design does not require any additional collateral from the operator. This comes at the cost of slightly changing the VTXO structure, having to involve the 1-of-n honest signer committee for every Ark transaction, and having to wait a period of 2Δ before finalising a payment.

The core idea behind this approach is to deter misbehaviour not by burning a large collateral, but to burn any VTXO that is being double-spent. If every honest receiver of an Ark transaction $\mathtt{tx}^{\mathtt{ark}}$ checks that

1) any collaborative spend of any input VTXO of $\mathtt{tx}^{\mathtt{ark}}$ can only occur a period t_p after the VTXO could be spent by the burn transaction,
2) for each input VTXO of $\mathtt{tx}^{\mathtt{ark}}$, if the VTXO has value v and locking script `vtxoLockScript`, $\mathtt{tx}^{\mathtt{ark}}$ has an output with value $v' > 0$ and locking script `vtxoLockScript`, and
3) no conflicting transaction has been broadcast by another user 2Δ after broadcasting the received $\mathtt{tx}^{\mathtt{ark}}$,

then the Ark transaction can safely be finalised. Indeed, we argue that in case of misbehaviour, this procedure ensures that only one honest user could be defrauded before the misbehaviour is detected by an honest user (who is online by (A4)), guaranteeing that the gain from misbehaving is smaller than the subsequent punishment.

By waiting for 2Δ after sending out $\texttt{tx}^{\texttt{ark}}$ (check 3), an honest receiver, which we call Bob, can be sure no other honest user has accepted a conflicting payment. Indeed, if Bob receives $\texttt{tx}^{\texttt{ark}}$ at time t and broadcasts it together with its path immediately, it can be sure that every honest user has received it by $t+\Delta$. Now, if a conflicting transaction is received by another honest user, say Charlie, after $t+\Delta$, the misbehaviour will be detected and Charlie will not accept the payment, meaning that the misbehaviour is detected. In the worst case only Bob will have been defrauded. If a conflicting transaction is received by Charlie before $t+\Delta$, Charlie also broadcast its transaction path to all honest users, meaning Bob receives it by $t+2\Delta$. At that point, Bob will not accept the payment (neither will Charlie, in fact), and misbehaviour will have been detected.

It is important to notice that at most one honest user can be defrauded. Say that the malicious sender spends an amount v in $\texttt{tx}^{\texttt{ark}}$. By check 2, an honest user will only accept that transaction if it receives an amount $v - v'$, and v' goes back to the sender. In other words, defrauding at most one honest user can only lead to a gain less than v since $v' > 0$. Since the misbehaviour will be detected, the input VTXOs of $\texttt{tx}^{\texttt{ark}}$ will be burnt (as the burn transaction has priority over any other VTXO spending path, by check 1), meaning that the net profit from misbehaving is $v - v' - v = -v' < 0$. As long as an honest user publishes the to-be-burned VTXO and the following burn transaction onchain, which is guaranteed by (A4), misbehaviour is thus disincentivised. Note that this come at an onchain cost, as the double-spent VTXO may be preceded by multiple Ark transactions. Further research may explore reward mechanisms to incentivise actually enforcing the punishment, possibly having dedicated entities (watchtowers), that broadcast the necessary transactions.

Appendix D. Commitment transactions and VTXTs

As an example, let us look at the commitment transaction created by our experiment in §6 for $n = 2$. The commitment transaction can be found on Mutinynet at: https://mutinynet.com/ by searching for the transaction ID `3fd6ddc344170c6ba90abb5a6ae7e4aaa1ca94c278de1501aba34fb7738740d3`. One can see one large input, as well as one large change output, together with a batch output of 10000 sat and a connector output of 660 sat. The connector output consists of two dust value (330 sat) outputs. The batch commits to two VTXOs, one of 8000 sat and one of 2000 sat. Below, we share the transactions in PSBT format for conciseness. These can easily be decoded into JSON via a website such as https://chainquery.com/bitcoin-cli/decodepsbt:

- VTXT Root, spending the batch output and having 2 P2TR outputs for the leaf transactions, and one anchor output: `cHNidP8BAJYDAAAAAdNAh3O3T6OrARXeeMKUyqGq5OdqWrsKqWsMF0TD3dY/AAAAAAD////A0AfAAAAAAAAIlEg1uuTP5t4SvcKMZtq4x3b6J5dADh8hC4t+79wnvdQBkzQBwAAAAAAACJRIHaEDwnIyDSOyHFf/jLuenc2dJ1lRTh4LnlMw7wVjN5lAAAAAAAAAAEUQJOcwAAAAAARNAyBga5bp6IDKv35ReC9Nqn0lDvB6Km2uitFQC88HNBkRvrSBQotq8004h1E9+pNUVM9VO6mg8q74vRJDwvnHZewxjb3NpZ25lcgAAAAAhAlWozfpWDEPyugdJ95dGUrYTUFjZVaqhCezPp8QKtyUSDGNvc2lnbmVyAAAAASEChNDc7qODfpNp/P7KaCotiCExEw6RXcLIgJkm1HtwFO4MY29zaWduZXIAAAACIQKbjuy0+nlO9A5YxcUwsNP7V/Tt56lculCTrhoZXbbWugZleHBpcnkDUztAAAAAAA==`
- Leaf 1, having the VTXO with value 8000 sat as output, together with an anchor: `cHNidP8BAGsDAAAAAQdO2R6SLbqa1+KtulJ84a/0qnMC5t0XNkzacdHcdVSTAAAAAAD/////AkAfAAAAAAAAIlEg7N6TSBPPzHXoOwvg7b5YxseoaDNxayBBQEmnhRvXLroAAAAAAAAAAARRAk5zAAAAAAABE0AYwVd0VW4mvU6RT1pdv+Nz1wAZ8nXmbxmsSooq2dj0VbHvmGe3sC7YzyXLEPhJ9PpLYnflBFMSSog/e/Cqy85pDGNvc2lnbmVyAAAAACECVajN+lYMQ/K6B0n3l0ZSthNQWNlVqqEJ7M+nxAq3JRIMY29zaWduZXIAAAABIQKE0Nzuo4N+k2n8/spoKi2IITETDpFdwsiAmSb1Ue3AU7gZleHBpcnkDUztAAAAA`
- Leaf 2, having the VTXO with value 2000 sat as output, together with an anchor : `cHNidP8BAGsDAAAAAQdO2R6SLbqa1+KtulJ84a/0qnMC5t0XNkzacdHcdVSTAQAAAAD/////AtAHAAAAAAAAIlEgk4Bo7yqsimoNGhA5XrjZnY0HpJU6dY9coXomK4rSHi8AAAAAAAAAAARRAk5zAAAAAAABE0ASijnyUYyN6+A+ab+qpb7WUmz8ngKWaLpBqBiVcYUTCjuNwkERVqaOIVtsr/iETZ00kpEdae02Pvzwt+Jt7ttiDGNvc2lnbmVyAAAAACECVajN+lYMQ/K6B0n3l0ZSthNQWNlVqqEJ7M+nxAq3JRIMY29zaWduZXIAAAABIQKbjuy0+nlO9A5YxcUwsNP7V/Tt56lculCTrhoZXbbWugZleHBpcnkDUztAAAAA`

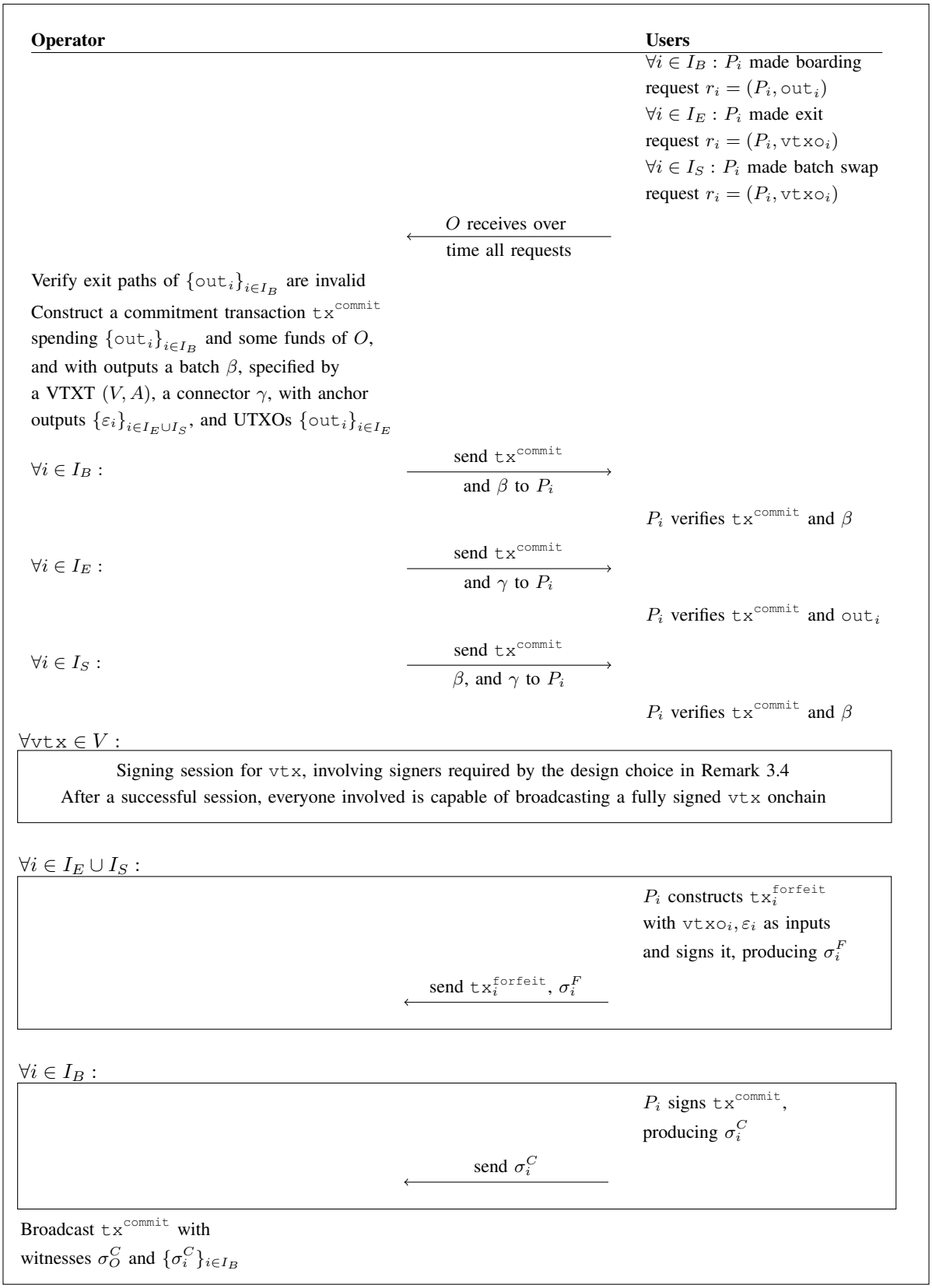

Figure A.1: High-level construction of a commitment transaction from requests indexed by I_B, I_E, I_S for boarding, exit, and batch swap request, respectively.

Appendix E.
Detailed Protocol Specification

We start off by listing some conventions and simplifying assumptions:

- We denote by $\texttt{batch}(\texttt{vtxo})$ the batch output containing $\texttt{vtxo}$.
- A party P can represents a single user or multiple users. We assume for simplicity that one entity communicates with the operator and can organise the users behind it to provide the necessary witnesses, and we assume that the view $\mathcal{L}^P$ of this entity is well-defined (for example that the multiple users represented by P all agree on the state of the blockchain).
- For any VTXO that has multiple spending conditions where different parties can provide witnesses satisfying one of these conditions, we assume that all these parties are aware of the state of this VTXO, i.e., whether for a given Ark state $\Sigma = (C, F, S)$, this VTXO is in C, F, or S (see Definition F.4. This is most relevant for the request routines (Routine 2, 4, 5, and 7), to ensure that honest requesters never make a request spending an already spent VTXO.

Furthermore, keep in mind that the lists $\texttt{vtxos}$ and $\texttt{vtxos}'$ should be understood here as lists of tuples $(\texttt{value}, \texttt{vtxoLockScript})$, such that O can read off all spending paths from $\texttt{vtxoLockScript}$. Of course, the locking script itself present in any transaction would only contain the tweaked public key committing to aforementioned spending paths.

The protocol setup is minimal and presented in Routine 1. It only requires the Operator O to let users know how they can create a boarding address, which boils down to O sharing its public key pk_O and some other Ark-specific parameters, for example on a website. Apart from that, a number of empty lists are instantiated for the operator to keep track of user requests and VTXOs. Finally, O defines a batching policy: a set of rules that specifies when and how O should construct a commitment transaction. This policy may also be published.

Users who want to board the Ark can now do so by sending a boarding request to the operator, as described in Routine 2. This boarding request specifies a list of VTXOs which the user in question wants to add to the Ark.

The Ark operator can now verify this boarding request, making sure that the UTXO specified in the request is not already spent, checking the locking script is of the correct form (in particular that there are no other spending paths), amongst others. This is specified in Routine 3.

We can now also specify the other types of request users can make to the operator: batch swap requests (Routine 4), exit requests (Routine 5), Ark transaction requests (Routine 7), and their respective verification procedures (Routines 8, 9 and 10). For the Ark transaction, we also write out the procedure to send an Ark transaction to the receiver (Routine 11), and what checks the receiver should subsequently perform (Routine 12).

Out of all of the received requests, the operator can select a number of them to construct a commitment transaction, as detailed in Routine 13. The batch outputs of this transaction are specified by a batch template, which is a list of VTXTs (one for each batch output). How these VTXTs are constructed exactly is specified in the batching policy from Routine 1 and is up to the operator. It employs its own rules and optimisations to construct the VTXTs that go beyond this protocol specification. Of course, the VTXTs are constructed such that the obtained batch outputs satisfy Definition 3.3. Similarly, the connector outputs are specified by a connector template: a list of connector outputs, and a connector function γ, mapping each VTXO that is being batch swapped or exited from to a leaf of one of the VTXTs in the connector template.

The commitment transaction produced by O is still missing signatures. O has to communicate with all the relevant parties, who will at least each individually check the commitment transaction is valid (Routines 14, 15 and 18). Furthermore, parties who sent a boarding request or batch swap request should make sure their requested VTXOs are included in the batch template, and parties who sent a batch swap or exit request should find the anchor output in the connector template associated with their old VTXOs that are being swapped or exited.

If every party convinced themselves that the commitment transaction is correct, parties who sent a batch swap or exit request will produce forfeit transactions (Routine 16) that are only valid once the commitment transaction holding the specific, corresponding anchor output is available onchain, and multiple signing sessions (Routine 6) will happen to produce witnesses for the commitment transaction itself, and for the virtual transactions contained in the different batch outputs.

All the steps to go from user requests to a fully signed commitment transaction are gathered in Routine 22. At the end, the commitment transaction can be broadcast, and the operator can update the various lists to keep track of the VTXOs and open requests. This is specified in Routine 21.

Finally, we specify the behaviour of honest parties, apart from how they should act when building commitment transactions. For the operator, this is done in Routine 23. For example, the operator has to monitor the chain for exited VTXOs and possibly respond by broadcasting the necessary reset and/or forfeit transactions. Other honest parties also have some tasks to perform, however not continuously. In Routine 24, we describe what a party wanting to spend a VTXO should do: first try to spend collaboratively, and if at some point this has still failed, exit unilaterally (Routine 19) and spend the created UTXO. In Routine 17, we give the steps the recipient of an Ark transaction should take.

Routine 1 Setup by Operator O with public key pk_O.

1: O publishes its public key pk_O, as well as the following Ark parameters:
- t_b: Timeout period for boarding transaction output.
- t_e: Minimum amount of blocks that should pass before a batch expires.
- t_v: Minimum delay in VTXO unilateral script path.
- t_r: Time after which if the commitment transaction is still not included the operator will retry.

2: O also instantiates the variables:
- `toBoard` := [ ], the list of all boarding requests that have been verified by O and can be added to a commitment transaction.
- `toBatchSwap` := [ ], the list of all batch swap requests that have been verified by O and can be added to a commitment transaction.
- `toExit` := [ ], the list of all exit requests that have been verified by O and can be added to a commitment transaction.
- `unconfirmed` := [ ], the list of all VTXOs that are part of the batch output of a transaction which has not yet been confirmed onchain.
- `confirmedVTXO` := [ ], the list of all VTXOs that are part of the batch output of a commitment transaction which has been confirmed onchain.
- `confirmedBatches` := [ ], the list of all batch outputs that are part of a commitment transaction which has been confirmed onchain.
- `expired` := [ ], the list all VTXOs that are part of a batch output of which the expiry timelock passed.
- `unconfirmedSpent` := [ ], the list of all VTXOs that have been spent, either by an Ark transaction, or by a batch swap or exit in an unconfirmed commitment transaction.
- `spent` := [ ], the list of all pairs $(\texttt{vtxo}, \texttt{tx})$ where `vtxo` could be spent onchain by a valid reset or forfeit transaction `tx`.
- `replaced` := [ ], the list of all VTXOs that are part of the batch output of a transaction which has been double-spent or invalidated.
- `preSpent` := [ ], the list of all VTXOs and boarding outputs spent in a yet unconfirmed boarding, batch swap, or exit request.
- `preConfirmed` := [ ], the list of all VTXOs that are output from Ark transactions, but not yet batch swapped.
- `unconfirmedBoardings` := [ ], `unconfirmedBatchSwaps` := [ ], and `unconfirmedExits` := [ ], the lists of all tuples of boarding, batch swap, and exit requests, respectively, with the fully signed, unconfirmed commitment transaction they have been processed in.
- `confirmedBoardings` := [ ], `confirmedBatchSwaps` := [ ], `confirmedExits` := [ ], the lists of all tuples of boarding, batch swap, and exit requests, respectively, with the fully signed, confirmed commitment transaction they have been processed in.

3: Finally, O specifies a *batching policy*. We abstract this policy away into a *batching black box* β, which deterministically maps a given state of the lists `toBoard`, `toBatchSwap` and `toExit` of available, verified requests to lists `boardings`, `batchSwaps` and `exits` of requests included in the commitment transaction, as well as templates `batchTmp`, `connectorTmp` and γ that specify how the VTXOs and anchor outputs will be arranged in batch and connector outputs of the commitment transaction. This policy may also include limits such as the maximum number of outputs per virtual transaction, or the maximum tree depth.

Note that β can also be a function of other external factors, such as the current time or onchain fees. For simplicity, we assume O shares its lists `toBoard`, `toBatchSwap` and `toExit` with β, listens to β and starts building a commitment transaction as soon as β outputs a tuple of the form $(\texttt{boardings}, \texttt{batchSwaps}, \texttt{exits}, \texttt{batchTmp}, \texttt{connectorTmp}, \gamma)$.

Routine 2 Boarding request to O for VTXO list $\texttt{vtxos}'$ with cosigner set N by a party P.

Require: A list of UTXOs $[\texttt{out}_j]_{j\in J} \subseteq \mathcal{L}^P_{-k}$ for which $\sum_{j\in J} \texttt{out}_j.\texttt{value} \geq v := \sum_{\texttt{vtxo}'\in\texttt{vtxos}'} \texttt{vtxo}'.\texttt{value}$, and such that for each $j \in J$, P can provide a witness w_j that allows $\texttt{out}_j$ to be spent for the boarding transaction. P should also know the operator public key pk_O. Finally, P should have sets S_C and S_U of collaborative and unilateral script paths, as well as an internal public key pk_I that is a NUMS (Nothing Up My Sleeve) point (such as the one recommended in BIP 341 [43]), in order for the key path to be unspendable. P should be able to provide witnesses for these script paths (collaborating with O when needed).

1: P constructs the boarding transaction $\texttt{tx}^{\texttt{board}}_P$, with $\texttt{tx}^{\texttt{board}}_P.\texttt{ins} = [\texttt{out}_j]_{j\in J}$, $\texttt{tx}^{\texttt{board}}_P.\texttt{wits} = [w_j]_{j\in J}$, and $\texttt{tx}^{\texttt{board}}_P.\texttt{outs} = \left[\texttt{out}^{\texttt{board}}_P\right]$, where $\texttt{out}^{\texttt{board}}_P.\texttt{value} = v$ and

$$\texttt{out}^{\texttt{board}}_P.\texttt{lockScript} = \texttt{Taproot}\big(\texttt{False}; S_C, S_U\big).$$

2: P submits $\texttt{tx}^{\texttt{board}}_P$ to Π_{BTC}.

3: Once $\texttt{tx}^{\texttt{board}}_P \in \mathcal{L}^P_{-k}$, P sends to O the boarding request

$$r = \big(\text{“boarding: ”}, P, N, \texttt{out}^{\texttt{board}}_P, pk_I, S_C, S_U, \texttt{vtxos}'\big).$$

Routine 3 **`verifyBoardingRequest(`**r**`):`** Verify boarding request $r = (\text{“boarding: ”}, P, N, \texttt{out}, pk_I, S_C, S_U, \texttt{vtxos}')$.

1: O checks that $\texttt{out} \in \mathcal{L}^O_{-k}$ and that $\texttt{out} \notin \texttt{preSpent}$.

2: O checks that each script path in S_C requires a signature from O, and that each script path in S_U has a relative timelock of at least t_b.

3: Given pk_I, S_C, S_U, O makes sure that the locking script of $\texttt{out}$ indeed commits to the correct script paths and unspendable key path.

4: O checks that $\texttt{out}.\texttt{value} \geq \sum_{\texttt{vtxo}'\in\texttt{vtxos}'} \texttt{vtxo}'.\texttt{value}$.

5: **for** $\texttt{vtxo}' \in \texttt{vtxos}'$ **do**

6: $\quad O$ checks that $\texttt{vtxo}'$ satisfies Definition 3.1.

7: **end for**

8: **if** all of the checks in Lines 1 - 7 are successful **then**

9: $\quad$ Add $(P, N, \texttt{out}, \texttt{vtxos}')$ to $\texttt{toBoard}$.

10: $\quad$ Add $\texttt{out}$ to $\texttt{preSpent}$.

11: **end if**

Routine 4 Batch swap request to O of a VTXO list $\texttt{vtxos}$ into a VTXO list $\texttt{vtxos}'$ with cosigner set N by P.

Require: VTXO lists $\texttt{vtxos} \subseteq C \cup F$ and $\texttt{vtxos}'$ such that for each $\texttt{vtxo} \in \texttt{vtxos}$, $\texttt{batch}(\texttt{vtxo})$ exists and is in $\mathcal{L}^P_{-k}$, $\texttt{vtxo}$ can be spent collaboratively by P and the operator, and such that $\sum_{\texttt{vtxo}\in\texttt{vtxos}} \texttt{vtxo}.\texttt{value} \geq \sum_{\texttt{vtxo}'\in\texttt{vtxos}'} \texttt{vtxo}'.\texttt{value}$.

1: P sends to O the batch swap request

$$r = (\text{“batch swap: ”}, P, N, \texttt{vtxos}, \texttt{vtxos}').$$

Routine 5 Exit request to O of a VTXO list $\texttt{vtxos}$ into a UTXO list $\texttt{utxos}$ by P.

Require: A VTXO list $\texttt{vtxos} \subseteq C \cup F$ such that for each $\texttt{vtxo} \in \texttt{vtxos}$, $\texttt{batch}(\texttt{vtxo})$ exists and is in $\mathcal{L}^P_{-k}$, and P is able to provide a witness to spend $\texttt{vtxo}$ via a collaborative VTXO script path. Also, a UTXO list $\texttt{utxos}$ such that $\sum_{\texttt{vtxo}\in\texttt{vtxos}} \texttt{vtxo}.\texttt{value} \geq \sum_{\texttt{out}\in\texttt{utxos}} \texttt{out}.\texttt{value}$.

1: P sends to O the exit request

$$r = (\text{“exit: ”}, P, \texttt{vtxos}, \texttt{utxos}).$$

Routine 6 **`musig(tx,`**N**`):`** Perform a $\texttt{MuSig2}$ signing session with public key set N for transaction $\texttt{tx}$.

1: The signers in N perform a $\texttt{MuSig2}$ signing session, such that after successful completion every signer has the signature of $\texttt{tx}$ under the aggregate public key $\bigoplus_{pk\in N} pk$.

Routine 7 Ark transaction request to O of a VTXO list $\texttt{vtxos} \subseteq C$ into a VTXO list $\texttt{vtxos}'$ by a party P.

Require: A VTXO list $\texttt{vtxos}$ such that for each $\texttt{vtxo} \in \texttt{vtxos}$, $\texttt{batch}(\texttt{vtxo})$ exists and is in $\mathcal{L}^P_{-k}$, and P is able to cooperate with the operator to provide a witness $w_{\texttt{vtxo}}$ to spend $\texttt{vtxo}$ via a collaborative VTXO script path $s_{C,\texttt{vtxo}}$. Also, a VTXO list $\texttt{vtxos}'$, such that $\sum_{\texttt{vtxo}\in\texttt{vtxos}} \texttt{vtxo.value} \geq \sum_{\texttt{vtxo}'\in\texttt{vtxos}'} \texttt{vtxo}'\texttt{.value}$.

P constructs for each $\texttt{vtxo} \in \texttt{vtxos}$ a reset transaction $\texttt{tx}^{\texttt{re}}_{P,\texttt{vtxo}}$, with $\texttt{tx}^{\texttt{re}}_{P,\texttt{vtxo}}\texttt{.ins} = \texttt{vtxos}$, $\texttt{tx}^{\texttt{re}}_{P,\texttt{vtxo}}\texttt{.wits} = [*]_{\texttt{vtxo}\in\texttt{vtxos}}$, and $\texttt{tx}^{\texttt{re}}_{P,\texttt{vtxo}}\texttt{.outs} = \texttt{out}^{\texttt{re}}_{\texttt{vtxo}}$, with $\texttt{out}^{\texttt{re}}_{\texttt{vtxo}}\texttt{.value} = \texttt{vtxo.value}$ and $\texttt{out}^{\texttt{re}}_{\texttt{vtxo}}\texttt{.lockScript} = \texttt{Taproot}(\texttt{False}; s_{C,\texttt{vtxo}}, \texttt{chkSig}_{pk_O} \wedge \texttt{absTlk}(T_{e,\texttt{vtxo}})))$, where $T_{e,\texttt{vtxo}}$ is the block height at which the batch containing $\texttt{vtxo}$ expires.

P constructs an Ark transaction $\texttt{tx}^{\texttt{ark}}_P$, with $\texttt{tx}^{\texttt{ark}}_P\texttt{.ins} = \left\{\texttt{out}^{\texttt{re}}_{\texttt{vtxo}}\right\}_{\texttt{vtxo}\in\texttt{vtxos}}$, $\texttt{tx}^{\texttt{ark}}_P\texttt{.wits} = [*]_{\texttt{vtxo}\in\texttt{vtxos}}$, and $\texttt{tx}^{\texttt{ark}}_P\texttt{.outs} = \texttt{vtxos}'$.

P sends to O the Ark transaction request

$$r = \left(\text{“Ark: ”}, P, \left\{\texttt{tx}^{\texttt{re}}_{P,\texttt{vtxo}}\right\}_{\texttt{vtxo}\in\texttt{vtxos}}, \texttt{tx}^{\texttt{ark}}_P, \texttt{vtxos}, \texttt{vtxos}'\right).$$

Routine 8 **`verifyBatchSwapRequest(`**r**`):`** Verify batch swap request $r = (\text{“batch swap: ”}, P, N, \texttt{vtxos}, \texttt{vtxos}')$.

1: **for** $\texttt{vtxo} \in \texttt{vtxos}$ **do**
2: O checks that $\texttt{vtxo} \in \texttt{confirmedVTXO} \cup \texttt{preConfirmed}$.
3: O checks that $\texttt{vtxo} \notin \texttt{preSpent}$.
4: **end for**
5: **for** $\texttt{vtxo}' \in \texttt{vtxos}'$ **do**
6: O makes sure that $\texttt{vtxo}'$ satisfies Definition 3.1.
7: **end for**
8: O checks that $\sum_{\texttt{vtxo}\in\texttt{vtxos}} \texttt{vtxo.value} \geq \sum_{\texttt{vtxo}'\in\texttt{vtxos}'} \texttt{vtxo}'\texttt{.value}$.
9: **if** all of the checks in Lines 1 - 8 are successful **then**
10: Add $(P, N, \texttt{vtxos}, \texttt{vtxos}')$ to $\texttt{toBatchSwap}$.
11: Add each $\texttt{vtxo} \in \texttt{vtxos}$ to $\texttt{preSpent}$.
12: **end if**

Routine 9 **`verifyExitRequest(`**r**`):`** Verify exit request $r = (\text{“exit: ”}, P, \texttt{vtxos}, \texttt{utxos})$.

1: **for** $\texttt{vtxo} \in \texttt{vtxos}$ **do**
2: O checks that $\texttt{vtxo} \in \texttt{confirmedVTXO} \cup \texttt{preConfirmed}$
3: O checks that $\texttt{vtxo} \notin \texttt{preSpent}$.
4: **end for**
5: O checks that $\sum_{\texttt{vtxo}\in\texttt{vtxos}} \texttt{vtxo.value} \geq \sum_{\texttt{out}\in\texttt{utxos}} \texttt{out.value}$.
6: **if** all of the checks in Lines 1 - 5 are successful **then**
7: Add $(P, \varnothing, \texttt{vtxos}, \texttt{utxos})$ to $\texttt{toExit}$.
8: Add each $\texttt{vtxo} \in \texttt{vtxos}$ to $\texttt{preSpent}$.
9: **end if**

Routine 10 **`verifyArkTxRequest(`**r**`):`** Verify Ark transaction request $r = \left(\text{“Ark: ”}, P, R, \texttt{tx}^{\texttt{ark}}, \texttt{vtxos}, \texttt{vtxos}'\right)$.

1: **for** $\texttt{vtxo} \in \texttt{vtxos}$ **do**
2: O checks that $\texttt{vtxo} \in \texttt{confirmedVTXO}$ and that $\texttt{vtxo} \notin \texttt{preSpent}$.
3: O verifies that there is a $\texttt{tx}^{\texttt{re}} \in R$ that can claim $\texttt{vtxo}$ at the corresponding batch's expiry.
4: O cooperates with P to produce a valid witness $w^{\texttt{ark}}_{\texttt{vtxo}}$ for each input of $\texttt{tx}^{\texttt{ark}}$.
5: **end for**
6: **for** $\texttt{vtxo}' \in \texttt{vtxos}'$ **do**
7: O makes sure that $\texttt{vtxo}'$ satisfies Definition 3.1.
8: **end for**
9: O checks that $\sum_{\texttt{vtxo}\in\texttt{vtxos}} \texttt{vtxo.value} \geq \sum_{\texttt{vtxo}'\in\texttt{vtxos}'} \texttt{vtxo}'\texttt{.value}$.
10: **if** all of the checks in Lines 1 - 9 are successful **then**
11: Add each $\texttt{vtxo}' \in \texttt{vtxos}'$ (as an output of $\texttt{tx}$) to $\texttt{preConfirmed}$.
12: Cooperate with P to create a witness $w^{\texttt{re}}_{\texttt{tx}}$ for each $\texttt{tx} \in R$.
13: Add $(\texttt{vtxos}, \texttt{tx})$ to $\texttt{spent}$, where $\texttt{tx} \in R$ is the corresponding fully signed reset transaction.
14: **end if**

Routine 11 `sendArkTx(`Q`,tx,vtxos′,paths):` Send an Ark transaction `tx` to Q by P.

Require: We assume `tx` is a fully signed Ark transaction, and that P has knowledge of the set `vtxos′` of actual locking scripts of `tx` that Q should verify, and `path(vtxo)` for each `vtxo` spent by `tx` (with the reset transaction $\texttt{tx}^{\texttt{re}}_{\texttt{vtxo}}$).

1: P sends to Q the fully signed Ark transaction `tx`, as well as the sets `vtxos′` of locking scripts of outputs of `tx` that Q should verify and $\texttt{paths} := \{\texttt{path}(\texttt{vtxo}) : \texttt{vtxo} \in \texttt{tx.ins}\}$.

Routine 12 `verifyArkTx(tx,vtxos′,paths):` Verify a received Ark transaction `tx` with subset of output locking scripts `vtxos′` and VTXT paths `paths` up to the transaction inputs by Q.

1: Q checks that the locking scripts in `vtxos′` are actually committed to by locking scripts of outputs of `tx`.
2: Q checks that if it were to submit all transactions in `paths` to Π_{BTC}, `tx` would be a valid Bitcoin transaction.

Routine 13 `commitmentTx(boardings, batchSwaps,exits,batchTmp,`
`connectorTmp,`γ**`):`** Construct commitment transaction with batch outputs according to `batchTmp`, connectors according to `connectorTmp` and γ, processing the boarding requests in `boardings`, batch swap requests in `batchSwaps`, and exit requests in `exits`.

Require: A list of verified boarding requests $\texttt{boardings} \subseteq \texttt{toBoard}$, a list of verified batch swap requests $\texttt{batchSwaps} \subseteq \texttt{toBatchSwap}$, a list of verified exit requests $\texttt{exits} \subseteq \texttt{toExit}$, a list of UTXOs $[\texttt{out}_j]_{j\in J} \subseteq \mathcal{L}^O_{-k}$ for which

$$\begin{aligned}\sum_{j\in J}\texttt{out}_j.\texttt{value} \geq & \sum_{(\cdot,\cdot,\cdot,\texttt{vtxos}')\in\texttt{boardings}} \sum_{\texttt{vtxo}'\in\texttt{vtxos}'} \texttt{vtxo.value} - \sum_{(\cdot,\cdot,\texttt{out},\cdot)\in\texttt{boardings}} \texttt{out.value} \\ & + \sum_{(\cdot,\cdot,\cdot,\texttt{vtxos}')\in\texttt{batchSwaps}} \sum_{\texttt{vtxo}'\in\texttt{vtxos}'} \texttt{vtxo}'.\texttt{value} + \sum_{(\cdot,\cdot,\texttt{vtxos},\cdot)\in\texttt{batchSwaps}} |\texttt{vtxos}|\cdot\varepsilon \\ & + \sum_{(\cdot,\cdot,\cdot,\texttt{utxos})\in\texttt{exits}} \sum_{\texttt{out}\in\texttt{utxos}} \texttt{out.value} + \sum_{(\cdot,\cdot,\texttt{vtxos},\cdot)\in\texttt{exits}} |\texttt{vtxos}|\cdot\varepsilon, \end{aligned} \tag{1}$$

and such that for each $j \in J$, $\texttt{out}_j$ can be spent by O with a witness w_j for the commitment transaction. Also, the collection of outputs of the leaves of all VTXTs in `batchTmp` should exactly coincide with $\bigcup_{(\cdot,\cdot,\cdot,\texttt{vtxos}')\in\texttt{boardings}} \texttt{vtxos}' \cup \bigcup_{(\cdot,\cdot,\cdot,\texttt{vtxos}')\in\texttt{batchSwaps}} \texttt{vtxos}'$, and `connectorTmp` should contain an anchor output for each VTXO in $\bigcup_{(\cdot,\cdot,\texttt{vtxos},\cdot)\in\texttt{batchSwaps}} \texttt{vtxos} \cup \bigcup_{(\cdot,\cdot,\texttt{vtxos},\cdot)\in\texttt{exits}} \texttt{vtxos}$. For ease of notation, given a VTXT `vtxt` in `batchTmp`, we define a function $\nu_{\texttt{vtxt}}$ which makes explicit for every $\texttt{tx} \in \texttt{vtxt}$ the cosigners that should sign this `tx` (note that this is already present implicitly in `batchTmp`).

1: Define h_O as the current block height according to the operator.
2: O constructs the unsigned commitment transaction $\texttt{tx}^{\texttt{commit}}$ with $\texttt{tx}^{\texttt{commit}}.\texttt{ins} = [\texttt{out}_j]_{j\in J} \cup \bigcup_{(\cdot,\cdot,\texttt{out},\cdot)\in\texttt{boardings}} \texttt{out}$, $\texttt{tx}^{\texttt{commit}}.\texttt{wits} = [*]_{j\in J} \cup \bigcup_{(\cdot,\cdot,\texttt{out},\cdot)\in\texttt{boardings}} [*]$ and $\texttt{tx}^{\texttt{commit}}.\texttt{outs}$ given by

$$\bigcup_{\texttt{vtxt}\in\texttt{batchTmp}} \texttt{out}^{\texttt{batch}}_{\texttt{vtxt}} \cup \bigcup_{\texttt{vtxt}\in\texttt{connectorTmp}} \texttt{out}^{\texttt{connector}}_{\texttt{vtxt}} \cup \bigcup_{(\cdot,\cdot,\cdot,\texttt{utxos})\in\texttt{exits}} \texttt{utxos},$$

where for each $\texttt{vtxt} \in \texttt{batchTmp}$, $\texttt{out}^{\texttt{batch}}_{\texttt{vtxt}}.\texttt{value} \geq \sum_{\texttt{tx}^{\texttt{leaf}}\in\texttt{leaves(vtxt)}} \texttt{tx}^{\texttt{leaf}}.\texttt{outs}[0].\texttt{value}$, where

$$\begin{aligned}\texttt{out}^{\texttt{batch}}_{\texttt{vtxt}}.\texttt{lockScript} = \texttt{Taproot}\big(&\texttt{False};\\ &\texttt{chkSig}_{pk_O} \wedge \texttt{absTlk}(h_O + 2k + t_e),\\ &\texttt{chkSig}_{pk_O \oplus \bigoplus_{pk\in\nu_{\texttt{vtxt}}(\texttt{root(vtxt)})} pk}\big),\end{aligned}$$

and where for each $\texttt{vtxt} \in \texttt{connectorTmp}$, $\texttt{out}^{\texttt{connector}}_{\texttt{vtxt}}.\texttt{value} \geq |\texttt{leaves(vtxt)}|\cdot\varepsilon$ and

$$\texttt{out}^{\texttt{connector}}_{\texttt{vtxt}}.\texttt{lockScript} = \texttt{Taproot}(\texttt{False}; \texttt{chkSig}_{pk_O}).$$

3: Let $\texttt{signerTmp} := [\,]$
4: **for** $\texttt{vtxt} = (V, A) \in \texttt{batchTmp}$ **do**
5: Let $V' := \{\nu_{\texttt{vtxt}}(\texttt{tx}) : \texttt{tx} \in V\}$.
6: Let $A' := \{(\nu_{\texttt{vtxt}}(\texttt{tx}), \nu_{\texttt{vtxt}}(\texttt{tx}^*)) \in V' : (\texttt{tx}, \texttt{tx}^*) \in A\}$.
7: Add (V', A') to `signerTmp`.
8: **end for**
9: **return** $(\texttt{tx}^{\texttt{commit}}, \texttt{signerTmp})$.

Routine 14 `verifyPath(`N`,`t`,`$\texttt{tx}^{\texttt{commit}}$`,vtxo,vtxt,st):` Verify using signer tree $\texttt{st} = (V', A')$ that VTXT $\texttt{vtxt} = (V, A)$ correctly includes $\texttt{vtxo}$ and enables unilateral exit controlled by N with expiry time t, and is properly included in $\texttt{tx}^{\texttt{commit}}$.

Require: We assume there is a unique bijection $\phi_V : V \to V'$ and a unique bijection $\phi_A : A \to A'$ such that $\phi_A((v_1, v_2)) = (\phi_V(v_1), \phi_V(v_2))$, otherwise, the routine trivially returns `False`. The operator public key pk_O should be known.

1: **if** $\exists \texttt{tx} \in \texttt{leaves}(\texttt{vtxt}) : \texttt{vtxo} \in \texttt{tx.outs}$. **then**
2: Write $(\texttt{tx}_1, \ldots, \texttt{tx}_\ell) := \texttt{path}(\texttt{vtxo})$.
3: **else**
4: **return** `False`.
5: **end if**
6: Write $\texttt{tx}_0 := \texttt{tx}^{\texttt{commit}}$.
7: Check $\texttt{tx}_\ell\texttt{.outs} = [\texttt{vtxo}]$.
8: **for** $i = \ell, \ldots, 1$ **do**
9: Check $\phi_V(\texttt{tx}_i) \supseteq N$.
10: Check $\exists \texttt{out} \in \texttt{tx}_{i-1}\texttt{.outs} : \texttt{tx}_i\texttt{.ins} = [\texttt{out}]$ and call it $\texttt{out}^\star$.
11: Check $\texttt{out}^\star\texttt{.value} \geq \sum_{\texttt{out}^\texttt{i} \in \texttt{tx}_i\texttt{.outs}} \texttt{out}^\texttt{i}\texttt{.value}$ and:

$$\texttt{out}^\star\texttt{.lockScript} = \texttt{Taproot}\big(\texttt{False};\ \texttt{chkSig}_{pk_O} \wedge \texttt{absTlk}(t),\ \texttt{chkSig}_{pk_O \oplus \bigoplus_{pk \in \phi_V(\texttt{tx}_i)} pk}\big).$$

12: **end for**
13: **if** all of the checks in Lines 1 - 12 are successful **then**
14: **return** $(\texttt{vtxt}, \texttt{out}^\star)$.
15: **end if**

Routine 15 `verifyConnector(vtxo,`$\texttt{tx}^{\texttt{commit}}$`,connectorTmp,`$\gamma$`):` Verify using signer tree $\texttt{st} = (V', A')$ that VTXT $\texttt{vtxt} = (V, A)$ correctly includes $\texttt{vtxo}$ and enables unilateral exit controlled by N with expiry time t, and is properly included in $\texttt{tx}^{\texttt{commit}}$.

1: **if** $\exists \texttt{vtxt} \in \texttt{connectorTmp} : \exists \texttt{tx} \in \texttt{leaves}(\texttt{vtxt}) : \gamma(\texttt{vtxo}) = \texttt{tx}$ **then**
2: Denote this $\texttt{vtxt}$ by $\texttt{vtxt}^*$ and $\texttt{tx}$ by $\texttt{tx}^*$.
3: Write $(\texttt{tx}_1, \ldots, \texttt{tx}_\ell) := \texttt{path}(\texttt{tx}^\star)$, i.e. the path of $\texttt{tx}^\star$ in $\texttt{vtxt}^*$.
4: Write $\texttt{tx}_0 := \texttt{tx}^{\texttt{commit}}$.
5: **for** $i = \ell, \ldots, 1$ **do**
6: Check $\exists \texttt{out} \in \texttt{tx}_{i-1}\texttt{.outs} : \texttt{tx}_i\texttt{.ins} = [\texttt{out}]$ and call it $\texttt{out}^\star$.
7: Check $\texttt{out}^\star\texttt{.value} \geq \sum_{\texttt{out}^\texttt{i} \in \texttt{tx}_i\texttt{.outs}} \texttt{out}^\texttt{i}\texttt{.value}$.
8: **end for**
9: **end if**
10: **if** all of the checks in Lines 1 - 8 are successful **then**
11: **return** `True`.
12: **end if**

Routine 16 `forfeitTx(vtxo,out):` Forfeit transaction for $\texttt{vtxo}$ using anchor output $\texttt{out}$ by P.

Require: It is assumed P is able to cooperate with O to produce a witness $w_{\texttt{vtxo}}$ to spend $\texttt{vtxo}$ via a collaborative VTXO script path, and that P knows the operator public key pk_O.

1: P constructs the transaction $\texttt{tx}^{\texttt{forfeit}}$, with $\texttt{tx}^{\texttt{forfeit}} = [\texttt{vtxo}, \texttt{out}]$, $\texttt{tx}^{\texttt{forfeit}}\texttt{.wits} = [*, *]$, and $\texttt{tx}^{\texttt{forfeit}}\texttt{.outs} = [(\texttt{vtxo.value} + \texttt{out.value}, \texttt{chkSig}_{pk_O})]$.
2: P sends $\texttt{tx}^{\texttt{forfeit}}$ to O.

Routine 17 Tasks a party P performs whenever receiving an Ark transaction.

Require: Assume P is the recipient of a $\texttt{sendArkTx}(P, \texttt{tx}, \texttt{vtxos}', \texttt{paths})$ operation, and that P is able to provide a witness to a collaborative script path of all VTXOs $\texttt{vtxos}'_P \subseteq \texttt{vtxos}'$.

1: **if** $\texttt{verifyArkTx}(\texttt{tx}, \texttt{vtxos}', \texttt{paths})$ **then**
2: Follow Routine 24 for $\texttt{vtxos}'_P$, with P batch swapping or exiting collaboratively.
3: **end if**

Routine 18 `verifyCommitTx(tx^commit, batchTmp, signerTmp, connectorTmp, γ)`: Verify commitment transaction $\texttt{tx}^{\texttt{commit}}$ with batch template `batchTmp`, signer template `signerTmp`, connector template `connectorTmp` and connector function γ by a party P.

Require: P is assumed to be involved in at least one request that is being processed by $\texttt{tx}^{\texttt{commit}}$, either as witness provider or cosigner. We respectively write by `boardings`, `batchSwaps`, `exits` the sets of boarding, batch swap, and exit requests included in $\texttt{tx}^{\texttt{commit}}$, for which P either made the request, or has its public key in the cosigner set. P should know the operator public key pk_O and the cosigner set N of public keys. All checks below are performed by P.

1: Check that $\sum_{\texttt{out} \in \texttt{tx}^{\texttt{commit}}.\texttt{ins}} \texttt{out.value} \geq \sum_{\texttt{out} \in \texttt{tx}^{\texttt{commit}}.\texttt{outs}} \texttt{out.value}$.
2: Let h_P be the current block height according to P and check that $h_O + 2k + t_e \geq h_P + 2k + t_e$.
3: Let $R_P := [\,]$.
4: **for** $(P, N, \texttt{out}, \texttt{vtxos}') \in \texttt{boardings}$ **do**
5: Check that $\texttt{out} \in \texttt{tx}^{\texttt{commit}}.\texttt{ins}$.
6: **for** $\texttt{vtxo}' \in \texttt{vtxos}'$ **do**
7: Check $\exists \texttt{vtxt} \in \texttt{batchTmp} : \exists \texttt{tx} \in \texttt{leaves}(\texttt{vtxt}) : \texttt{vtxo}' \in \texttt{tx.outs}$.
8: Denote `vtxt` by $\texttt{vtxt}^*$ and the corresponding tree in `signerTmp` by $\texttt{st}^*$.
9: Let $\texttt{res} := \texttt{verifyPath}(P, N, h_O + 2k + t_e, \texttt{tx}^{\texttt{commit}}, \texttt{vtxo}', \texttt{vtxt}^*, \texttt{st}^*)$.
10: **if** $\texttt{res} = (\texttt{vtxt}^*, \texttt{out}^*)$ for some $\texttt{out}^* \in \texttt{tx}^{\texttt{commit}}.\texttt{outs}$ **then**
11: Add $(\texttt{vtxt}^*, \texttt{out}^*)$ to R_P.
12: **else**
13: **return** `False`
14: **end if**
15: **end for**
16: **end for**
17: **for** $(P, N, \texttt{vtxos}, \texttt{vtxos}') \in \texttt{batchSwaps}$ **do**
18: **for** $\texttt{vtxo} \in \texttt{vtxos}$ **do**
19: $\texttt{verifyConnector}(\texttt{vtxo}, \texttt{tx}^{\texttt{commit}}, \texttt{connectorTmp}, \gamma)$.
20: **end for**
21: **for** $\texttt{vtxo}' \in \texttt{vtxos}'$ **do**
22: Check $\exists \texttt{vtxt} \in \texttt{batchTmp} : \exists \texttt{tx} \in \texttt{leaves}(\texttt{vtxt}) : \texttt{vtxo}' \in \texttt{tx.outs}$.
23: Let $\texttt{vtxt}^* := \texttt{vtxt}$ and the corresponding tree in `signerTmp` be $\texttt{st}^*$.
24: Let $\texttt{res} := \texttt{verifyPath}(P, N, h_O + 2k + t_e, \texttt{tx}^{\texttt{commit}}, \texttt{vtxo}, \texttt{vtxt}^*, \texttt{st}^*)$.
25: **if** $\texttt{res} = (\texttt{vtxt}^*, \texttt{out}^*)$ for some $\texttt{out}^* \in \texttt{tx}^{\texttt{commit}}.\texttt{outs}$ **then**
26: Add $(\texttt{vtxt}^*, \texttt{out}^*)$ to R_P.
27: **else**
28: **return** `False`
29: **end if**
30: **end for**
31: **end for**
32: **for** $(P, N, \texttt{vtxos}, \texttt{utxos}) \in \texttt{exits}$ **do**
33: **for** $\texttt{vtxo} \in \texttt{vtxos}$ **do**
34: $\texttt{verifyConnector}(\texttt{vtxo}, \texttt{tx}^{\texttt{commit}}, \texttt{connectorTmp}, \gamma)$.
35: **end for**
36: **for** $\texttt{out} \in \texttt{utxos}$ **do**
37: Check that $\texttt{out} \in \texttt{tx}^{\texttt{commit}}.\texttt{outs}$ and add $(\texttt{out}, \texttt{out})$ to R_P.
38: **end for**
39: **end for**
40: Check there are no $(x_1, y_1), (x_2, y_2) \in R_P$ for which $x_1 = x_2$ but $y_1 \neq y_2$.
41: Check any additional requirements announced by O (such as maximum number of outputs per virtual transaction, maximum tree depth, or the presence of always spendable anchor outputs $(\varepsilon, \texttt{True})$) by inspecting each VTXT in `batchTmp`.
42: **if** all of the checks in Lines 1 - 40 are successful **then**
43: **return** `True`.
44: **end if**

Routine 19 `unilateralExit(vtxo)`: Redeem a VTXO onchain via a unilateral exit by a party P.

Require: A VTXO `vtxo` such that `batch(vtxo)` exists and is in $\mathcal{L}^P_{-k}$. P also needs to possess all, fully signed, virtual transactions in `path(vtxo)` as defined in Definition 3.2.

P submits all transactions in `path(vtxo)` $\setminus \mathcal{L}^P_{-k}$ to Π_{BTC}.

Routine 20 `sweep(out)`: Reclaim the remaining funds from an output `out`.

Require: We assume `out` has a Taproot script path of the form $\texttt{chkSig}_{pk_O} \wedge \texttt{absTlk}(t)$ for some block height t.

if `out` is spent by some $\texttt{tx}^\star \in \mathcal{L}^O$ **then**
  for $\texttt{out}^\star \in \texttt{tx}^\star\texttt{.outs}$ **do**
    if $\texttt{out}^\star$ has a Taproot script path $\texttt{chkSig}_{pk_O} \wedge \texttt{absTlk}(t)$ for some t **then**
      `sweep(out`*`)`.
    end if
  end for
else
  Construct `tx` spending `out` with output $(v, \texttt{chkSig}_{pk_O})$, where $v \leq \texttt{out.value}$.
  Submit `tx` to Π_{BTC}.
end if

Routine 21 `submitCommitTx(tx`$^{\texttt{commit}}$`,boardings, batchSwaps, exits, old, new)`: Update all the required request and VTXO lists after submitting a fully signed commitment transaction $\texttt{tx}^{\texttt{commit}}$ constructed from `boardings`, `batchSwaps` and `exits`, yielding spent and created VTXO sets `old` (paired with the corresponding forfeit transaction) and `new`. All operations are done by O.

Submit $\texttt{tx}^{\texttt{commit}}$ to Π_{BTC}, denote by h the block height at which this happens according to O.
Remove all VTXOs in `old` from `confirmedVTXO` and `preConfirmed` and add $(\texttt{tx}^{\texttt{commit}}, \texttt{old})$ to `unconfirmedSpent`.
Add $(\texttt{tx}^{\texttt{commit}}, \texttt{new})$ to `unconfirmed`.
Remove `boardings` from `toBoard` and add $(\texttt{tx}^{\texttt{commit}}, \texttt{boardings})$ to `unconfirmedBoardings`.
Remove `batchSwaps` from `toBatchSwap` and add $(\texttt{tx}^{\texttt{commit}}, \texttt{batchSwaps})$ to `unconfirmedBatchSwaps`.
Remove `exits` from `toExit` and add $(\texttt{tx}^{\texttt{commit}}, \texttt{exits})$ to `unconfirmedExits`.
if $\texttt{tx}^{\texttt{commit}} \in \mathcal{L}^O_{-k}$ **then**
  Remove $(\texttt{tx}^{\texttt{commit}}, \texttt{old})$ from `unconfirmedSpent`.
  Add all pairs of VTXOs and forfeit transactions in `old` to `spent`.
  Remove $(\texttt{tx}^{\texttt{commit}}, \texttt{new})$ from `unconfirmed`.
  Add all VTXOs in `new` to `confirmedVTXO`.
  Add all batch outputs of $\texttt{tx}^{\texttt{commit}}$ to `confirmedBatches`.
  Remove $(\texttt{tx}^{\texttt{commit}}, \texttt{boardings})$ from `unconfirmedBoardings`.
  Remove $(\texttt{tx}^{\texttt{commit}}, \texttt{batchSwaps})$ from `unconfirmedBatchSwaps`.
  Remove $(\texttt{tx}^{\texttt{commit}}, \texttt{exits})$ from `unconfirmedExits`.
  Add $(\texttt{tx}^{\texttt{commit}}, \texttt{boardings})$ to `confirmedBoardings`.
  Add $(\texttt{tx}^{\texttt{commit}}, \texttt{batchSwaps})$ to `confirmedBatchSwaps`.
  Add $(\texttt{tx}^{\texttt{commit}}, \texttt{exits})$ to `confirmedExits`.
else if $\texttt{tx}^{\texttt{commit}}$ has never been in $\mathcal{L}^O_{-k}$ by block height $h + t_r$ **then**
  Remove $(\texttt{tx}^{\texttt{commit}}, \texttt{old})$ from `unconfirmedSpent`.
  Remove $(\texttt{tx}^{\texttt{commit}}, \texttt{new})$ from `unconfirmed`.
  Remove $(\texttt{tx}^{\texttt{commit}}, \texttt{boardings})$ from `unconfirmedBoardings`.
  Remove $(\texttt{tx}^{\texttt{commit}}, \texttt{batchSwaps})$ from `unconfirmedBatchSwaps`.
  Remove $(\texttt{tx}^{\texttt{commit}}, \texttt{exits})$ from `unconfirmedExits`.
  Add `boardings` to `toBoard`.
  Add `batchSwaps` to `toBatchSwap`.
  Add `exits` to `toExit`.
end if

Routine 22 `processRequests(boardings,batchSwaps,exits,batchTmp, connectorTmp,`γ`):` Construct the corresponding commitment transaction according to Routine 13 and finalise it by gathering the required signatures.

Require: It is assumed that the O and all cosigners represented in the requests `boardings`, `batchSwaps` and `exits` can communicate with each other. For ease of notation, we refer to the cosigners by their public keys, and we denote by $\mathcal{N}$ the set of all cosigners involved in all the requests processed here, and we denote by $\mathcal{P}$ the set of all requesters who made requests processed here.

O constructs $(\texttt{tx}^{\texttt{commit}}, \texttt{signerTmp}) := \texttt{commitmentTx}(\texttt{boardings}, \texttt{batchSwaps}, \texttt{exits}, \texttt{batchTmp}, \texttt{connectorTmp}, \gamma)$.
Let $\texttt{new} := \bigcup_{(\cdot,\cdot,\cdot,\texttt{vtxos}') \in \texttt{boardingsUbatchSwaps}} \texttt{vtxos}'$ be all to be created VTXOs.
Let $\texttt{old} := [\,]$.
for $S \in \mathcal{P} \cup \mathcal{N}$ **do**
    O sends to S the tuple $(\texttt{tx}^{\texttt{commit}}, \texttt{batchTmp}, \texttt{signerTmp}, \texttt{connectorTmp}, \gamma)$.
    if not $\texttt{verifyCommitTx}(\texttt{tx}^{\texttt{commit}}, \texttt{batchTmp}, \texttt{signerTmp}, \texttt{connectorTmp}, \gamma)$ **then**
        S aborts.
    end if
end for
Since $\texttt{verifyCommitTx}(\ldots)$ succeeded for everyone, there is for each $\texttt{vtxt} = (V, A) \in \texttt{batchTmp}$ a corresponding tree $\texttt{st} \in \texttt{signerTmp}$ with bijections $\phi_V^{\texttt{vtxt}} : V \to V'$ and $\phi_A^{\texttt{vtxt}} : A \to A'$ such that $\phi_A^{\texttt{vtxt}}((v_1, v_2)) = (\phi_V^{\texttt{vtxt}}(v_1), \phi_V^{\texttt{vtxt}}(v_2))$.
for $\texttt{vtxt} := (V, A) \in \texttt{batchTmp}$ **do**
    for $\texttt{tx} \in V$ **do**
        $\texttt{musig}(\texttt{tx}, \phi_V^{\texttt{vtxt}}(\texttt{tx}) \cup O)$.
    end for
end for
for $(P, N, \texttt{out}, \texttt{vtxos}') \in \texttt{boardings}$ **do**
    $\texttt{musig}(\texttt{tx}^{\texttt{commit}}, P \cup \{O\})$.
end for
for $(P, N, \texttt{vtxos}, \texttt{vtxos}') \in \texttt{batchSwaps}$ **do**
    for $\texttt{vtxo} \in \texttt{vtxos}$ **do**
        $\texttt{forfeitTx}(\texttt{vtxo}, \gamma(\texttt{vtxo}))$ yielding $\texttt{tx}_{\texttt{vtxo}}^{\texttt{forfeit}}$.
        O checks that $\texttt{tx}_{\texttt{vtxo}}^{\texttt{forfeit}}$ has:
        - inputs spending $\texttt{vtxo}$ and $\gamma(\texttt{vtxo})$.
        - one output sending all the funds to O.

        P and O cooperate to provide a collaborative witness to spend $\texttt{vtxo}$.
        O adds $(\texttt{vtxo}, \texttt{tx}_{\texttt{vtxo}}^{\texttt{forfeit}})$ to $\texttt{old}$, with $\texttt{tx}_{\texttt{vtxo}}^{\texttt{forfeit}}$ now fully signed.
    end for
end for
for $(P, \varnothing, \texttt{vtxos}, \texttt{utxos}) \in \texttt{exits}$ **do**
    for $\texttt{vtxo} \in \texttt{vtxos}$ **do**
        $\texttt{forfeitTx}(\texttt{vtxo}, \gamma(\texttt{vtxo}))$ yielding $\texttt{tx}_{\texttt{vtxo}}^{\texttt{forfeit}}$.
        O checks that $\texttt{tx}_{\texttt{vtxo}}^{\texttt{forfeit}}$ has:
        - inputs spending $\texttt{vtxo}$ and $\gamma(\texttt{vtxo})$.
        - one output sending all the funds to O.

        P and O cooperate to provide a collaborative witness to spend $\texttt{vtxo}$.
        O adds $(\texttt{vtxo}, \texttt{tx}_{\texttt{vtxo}}^{\texttt{forfeit}})$ to $\texttt{old}$, with $\texttt{tx}_{\texttt{vtxo}}^{\texttt{forfeit}}$ now fully signed.
        O adds $(\texttt{vtxo}, \texttt{tx}_{\texttt{vtxo}}^{\texttt{forfeit}})$ to $\texttt{old}$.
    end for
end for
Once O received a forfeit transaction for every $\texttt{vtxo}$ spent in a batch swap or exit request, O provides the witnesses $[w_j]_{j \in J}$ for the outputs $[\texttt{out}_j]_{j \in J}$.
$\texttt{submitCommitTx}(\texttt{tx}^{\texttt{commit}}, \texttt{boardings}, \texttt{batchSwaps}, \texttt{exits}, \texttt{old}, \texttt{new})$.

Routine 23 Tasks an operator performs continuously.

while `True` **do**
  if O receives a request $r = (\text{“boarding: ”}, P, N, \texttt{out}, pk_I, S_C, S_U, \texttt{vtxos}')$ **then**
    `verifyBoardingRequest`(r).
  end if
  if O receives a request $r = (\text{“Ark: ”}, P, \texttt{tx}, \texttt{vtxos}, \texttt{vtxos}')$ **then**
    `verifyArkTxRequest`(r).
  end if
  if O receives a request $r = (\text{“batch swap: ”}, P, N, \texttt{vtxos}, \texttt{vtxos}')$ **then**
    `verifyBatchSwapRequest`(r).
  end if
  if O receives a request $r = (\text{“exit: ”}, P, N, \texttt{vtxos}, \texttt{utxos})$ **then**
    `verifyExitRequest`(r).
  end if
  if β outputs $(\texttt{boardings}, \texttt{batchSwaps}, \texttt{exits}, \texttt{batchTmp}, \texttt{connectorTmp}, \gamma)$ **then**
    `processRequests`$(\texttt{boardings}, \texttt{batchSwaps}, \texttt{exits}, \texttt{batchTmp}, \texttt{connectorTmp}, \gamma)$.
  end if
  for $out \in$ `confirmedBatches` **do**
    if Current block height is past `out`'s batch expiry height **then**
      `sweep`(`out`).
    end if
  end for
  for $(\texttt{vtxo}, \texttt{tx}) \in$ `spent` **do**
    if $\texttt{vtxo} \in \mathcal{L}^O$ **then**
      Submit `tx` to Π_{BTC}.
    end if
  end for
end while

Routine 24 Tasks a party P performs in order to spend a set of VTXO `vtxos` P would want to spend collaboratively/unilaterally.

Require: Assume P is able to provide a witness to both a unilateral and collaborative script path of each VTXO in `vtxos`, and that P has knowledge of the fully signed virtual transactions on `path`(`vtxo`) for all $\texttt{vtxo} \in \texttt{vtxos}$. Let t be the batch expiry height. In the case of an Ark transaction, we assume that the transaction inputs can be accessed via the set `paths` of virtual transaction paths, that the transaction outputs are in the set $\texttt{vtxos}'$, and that P has defined a set of cosigner N and a function $\eta : N \to \mathcal{P}(\texttt{vtxos}')$ which maps each cosigner to a set of VTXOs that that cosigner should be able to batch swap or exit with.

if P's current block height is smaller than $t - 2k - 1$ **then**
  if P wants to perform a batch swap or exit **then**
    Initiate the desired batch swap or exit request.
    Collaborate with O in the corresponding `processRequests`$(\ldots)$.
  else if P wants to perform an Ark transaction `tx` **then**
    Initiate the desired Ark transaction request r with outputs $\texttt{vtxos}'$ according to Routine 7.
    if P obtains the fully signed Ark and reset transactions after a successful `verifyArkTxRequest`(r) **then**
      for $Q \in N$ **do**
        P executes `sendArkTx`$(Q, \texttt{tx}, \eta(Q), \texttt{paths})$.
      end for
    end if
  end if
  Perform the desired batch swap or exit request, collaborating with O.
else if P's current block height is $t - 2k - 1$ and spending `vtxos` failed **then**
  for $\texttt{vtxo} \in \texttt{vtxos}$ **do**
    P executes `unilateralExit`(`vtxo`) and spends the newly created UTXO.
  end for
end if

Appendix F. Formal Analysis

In this section, we formally define and prove the security of the Ark protocol. As mentioned in §2.1, we work with the Bitcoin backbone static model (fixed set of miners, static difficulty) presented in [37] to model the Bitcoin blockchain. In this model, we proceed in discrete rounds. By the synchrony assumption, every message sent at round r by an honest party will be received by all other honest party at round $r+1$. At round r, each Ark party Q maintains its own view of the ledger $\mathcal{L}^Q(r)$. We write $\mathcal{L}^Q_{-k}(r)$ for this local view with the last k blocks removed. We write $\mathcal{L}^1 \preceq \mathcal{L}^2$ if $\mathcal{L}^1$ is a prefix of $\mathcal{L}^2$. To reason about the confirmation of transactions and the possibility of reorgs, we treat Bitcoin as a protocol Π_{BTC} with depth parameter k and wait parameter u, which ensures, with overwhelming probability (w.o.p.),

- Persistence: For any two honest parties Q and Q', and any round r, we either have $\mathcal{L}^Q_{-k}(r) \preceq \mathcal{L}^{Q'}_{-k}(r)$ or $\mathcal{L}^Q_{-k}(r) \succeq \mathcal{L}^{Q'}_{-k}(r)$. We refer to $\mathcal{L}^Q_{-k}(r)$ as the *stable* part of Q's ledger, as this will no longer change from round r onward.
- Liveness: For any transaction $\mathtt{tx}$ submitted to Π_{BTC} by an honest party at round r, we have $\mathtt{tx} \in \mathcal{L}^Q_{-k}(s)$ for every honest Q by round $s = r + u$.

If we treat Bitcoin as above, one can show the following.

Corollary F.1 (Corollary 4.12 in [37])**.** *W.o.p.,* Π_{BTC} *ensures any* k *consecutive blocks in the chain of an honest party contain at least one block produced by an honest miner, who includes* all *valid transactions it has heard of by the round it produces the block.*

We translate the liveness property defined above to a statement in terms of rounds. To this end, we state and prove the following Lemma (adapted from Lemma 16 in [44]).

Lemma F.2. *If an honest party* Q *submits a transaction* $\mathtt{tx}$ *to* Π_{BTC} *at round* r*, at which time the latest stable block in* Q*'s ledger is* B^Q*, a block* B^* *including* $\mathtt{tx}$ *it will be in the stable part of the ledger of all honest parties at most* $3k$ *blocks away from* B^Q*, w.o.p.*

Proof. We know by Persistence that at round r, there are k consecutive blocks on top of B^Q in $\mathcal{L}^Q$. By Liveness, any $\mathtt{tx}$ submitted by an honest party Q to Π_{BTC} at round r will be in the stable part of the ledger of any honest party by round $s = r+u$. We now know two things: (i) every honest miner will have heard of $\mathtt{tx}$ from round $r+1$ onward. Therefore, $\mathtt{tx}$ will be included in the first honest block mined from round $r+1$ onward. However, by Corollary F.1, we know that w.o.p., there cannot be more than $k-1$ blocks between round r and round s. Indeed, if there would, this would mean there would be a sequence of k consecutive blocks that do not contain an honest block, as an honest block would have included $\mathtt{tx}$. Hence, there are at most $2k-1$ blocks between B^Q and B^*. (ii) by round s, B^* is in the stable part of every honest party's ledger, i.e., B^* is at least k blocks deep. Thus, combining (i) and (ii), B^* is in the stable part of every honest party's ledger after at most $3k$ consecutive blocks from B^Q. □

This implies, if an honest party Q's local view is currently at block height h, and Q submits $\mathtt{tx}$ to Π_{BTC} at that height, then w.o.p., $\mathtt{tx}$ will be included in a block that is a most $3k$ blocks away from Q's latest stable block (at block height $h-k$). That is, $\mathtt{tx}$ will be in the stable part of any honest party's ledger at latest at block height $h-k+3k = h+2k$. We use this realisation throughout the rest of the proofs and state it as a Corollary.

Corollary F.3. *If an honest party submits* $\mathtt{tx}$ *to* Π_{BTC} *at block height* h*,* $\mathtt{tx}$ *will be in the stable part of every honest party's ledger at latest at block height* $h+2k$*, w.o.p.*

We can now introduce the notion of an *Ark state*, which represents all VTXOs in an Ark. Next, we define *Ark state transitions* as sets of *Ark actions*. These notions will help us to formally define the security properties stated in §2.3.

Definition F.4 (Ark State)**.** *The* Ark state *of an Ark run by* O *is defined as a tuple* $\Sigma := (C, F, S)$*, where* C *is the set containing each VTXO, that is part of a batch confirmed onchain, for which there is no valid transaction spending that VTXO collaboratively in case it would be present onchain,* F *is the set of VTXOs that are outputs of Ark transactions, and* S *is the set of VTXOs for which* O *holds a fully signed reset transaction, or a fully signed forfeit transaction spending an anchor output confirmed onchain.*

Definition F.5 (Ark Action)**.** *An* Ark action *is a concise way to denote an Ark protocol action. It is a tuple* $\alpha = (P, N, O, v_{in}, u_{in}, v_{out}, u_{out}, a)$*, where* P *is the party providing the required witnesses to spend the VTXOs in* v_{in} *and UTXOs in* u_{in}*, in order to create the VTXOs in* v_{out} *and UTXOs in* u_{out}*.* N *is the cosigner set involved in committing any VTXO to a batch. Finally,* a *is a binary variable equal to 1 if and only if* α *is an Ark transaction.*

Definition F.6 (Ark State Transition)**.** *An* Ark state transition τ *is a set of Ark actions that changes the Ark state from* (C, F, S) *to* (C', F', S')*. We may also write* $(C, F, S) \xrightarrow{\tau} (C', F', S')$*.*

Ark state transitions include commitment transactions, unilateral exits, Ark transactions, and batch sweeps. For all but Ark transactions, these transitions only occur if the relevant transactions are confirmed onchain. While the latter three involve a single action each, a commitment transaction may group multiple actions. Thus, we may represent a commitment transaction $\mathtt{tx}$ as a set $\{\alpha_i\}_{i \in I}$ for some index set I. Conversely, a set of actions $\{\alpha_i\} i \in I$ with $\alpha_i = (P_i, N_i, v_{in,i}, u_{in,i}, v_{out,i}, u_{out,i}, 0)$ defines a commitment transaction $\mathtt{tx}$ with inputs $\bigcup_{i \in I} u_{in,i}$, outputs $\bigcup_{i \in I} u_{out,i}$, batch outputs holding $\bigcup_{i \in I} v_{out,i}$, requiring signatures from $\bigcup_{i \in I} N_i$, and connector outputs for $\bigcup_{i \in I} v_{in,i}$. Witnesses come from the corresponding parties in $\bigcup_{i \in I} P_i$.

As claimed in §2.3, the Ark protocol with operator O, $\Pi_{\mathsf{ark}}(O)$, gives a number of security guarantees. Informally, VTXO Security gives *static* guarantees that VTXOs cannot

be spent by unauthorised parties, and that authorised parties can always spend VTXOs up to a certain time. This is fundamental in showing User Balance Security. Ark Atomicity ensures that for any honest intent to change the state, the state either changes exactly according to the intent, or not at all. In both cases, honest users can rely on the guarantees of VTXO Security. Intuitively, Ark Atomicity ensures that transacting VTXOs works as expected. Finally, of most importance to an honest operator is that it does not lose funds. We phrase this as Operator Balance Security. Let us now precisely state each security property.

Theorem F.7 (VTXO Security)**.** *The protocol $\Pi_{\mathsf{ark}}(O)$ is*

(i) VTXO-safe*, i.e., for each Ark state $\Sigma = (C, F, S)$, for each $\texttt{vtxo} \in C$ confirmed in a batch with expiry T, and for each Ark state transition τ before T such that $(C, F, S) \xrightarrow{\tau} (C', F', S')$, if the cosigner set N from the action creating $\texttt{vtxo}$ is honest, we have that if $\texttt{vtxo} \in C$ and $\texttt{vtxo} \notin C'$, there must be an action $\alpha = (P, N', O, v_{in}, u_{in}, v_{out}, u_{out}, a) \in \tau$ such that α is a unilateral exit of $\texttt{vtxo}$, or P can construct with O a valid witness for $\texttt{vtxo}$.*

(ii) VTXO-live*, i.e., for each Ark state $\Sigma = (C, F, S)$, for each $\texttt{vtxo} \in C$ in a confirmed batch with expiry T, if the cosigner set N from the action creating $\texttt{vtxo}$ is honest, $\texttt{vtxo}$ can, at latest at block height $T - 2k - 1$, be turned w.o.p. into a UTXO with the same spending conditions by anyone who can submit the fully signed transactions in $\texttt{path}(\texttt{vtxo})$.*

Proof. Consider an arbitrary Ark state $\Sigma = (C, F, S)$, and let $\texttt{vtxo} \in C$ be a VTXO in a confirmed batch with expiry T (we assume that $C \neq \varnothing$). We assume that $\texttt{vtxo}$ has been created by an Ark action with honest cosigner set N. Routine 14 guarantees that N is among the signers required to spend the batch output before expiry, and every output on the path of virtual transactions to $\texttt{vtxo}$. Since N is honest, it will not agree to sign off on another transaction that would render $\texttt{vtxo}$ invalid. This realisation implies the following:

(i) For each state transition τ such that $(C, F, S) \xrightarrow{\tau} (C', F', S')$ before T, assume that $\texttt{vtxo} \in C$ and $\texttt{vtxo} \notin C'$. Since the batch containing $\texttt{vtxo}$ did not yet expire, the only way $\texttt{vtxo}$ would not be a part of C' any more, is if $\texttt{vtxo}$ would be turned into a UTXO via a unilateral exit, or if there exists a valid (virtual) transaction spending $\texttt{vtxo}$ collaboratively. This transaction can only exist if it includes a valid witness for $\texttt{vtxo}$'s collaborative locking scripts. P should thus be able to provide such a witness, cooperating with O.

(ii) Before batch expiry, the only way to spend the batch is with presigned transactions from the VTXT specifying the batch. Anyone in possession of the virtual transactions on $\texttt{path}(\texttt{vtxo})$ can thus submit these transactions, and be sure that there can be no double-spend. This as long as all transactions in the path get confirmed before block height T. By Corollary F.3, this happens w.o.p. within at most $2k$ blocks. In other words, as long as all transactions in $\texttt{path}(\texttt{vtxo})$ get submitted to Π_{BTC} by a party at latest at block height $T - 2k - 1$ in that party's local view, $\texttt{vtxo}$ will be available onchain as a confirmed UTXO.

□

Remark F.8. *Note how our formal model deviates from reality in the previous proof. For one, we assume that all transactions in $\texttt{path}(\texttt{vtxo})$ are submitted at once, and can all be included at once in a block. While this is true if the party submitting the transactions would talk directly to a miner, it is not true in practice with Bitcoin's current P2P relay policy. Bitcoin Core at the time of writing only supports packages of one parent and one child transaction being relayed through the network. Hence, a chain of transactions like $\texttt{path}(\texttt{vtxo})$, which relies on CPFP (child-pays-for-parent) in order to bump the fees of the parent transactions to guarantee inclusion onchain, may not get relayed reliably through the network. One would have to first make sure that parent transaction are successfully broadcast and included in the mempools of miners, and then broadcast the children (one-by-one). At the time of writing, work is ongoing [45] to introduce* full package relay *into Bitcoin, which would remove this inconvenience as it would allow larger packages of transactions to be relayed together.*

Second, the Bitcoin backbone model effectively assumes arbitrarily large blocks, such that all transactions submitted (and available in an honest miner's mempool) will be included by that honest miner in one single block. This assumption may no longer be realistic in case of a bank run scenario (see §7), where congestion may occur by the sheer number of transactions submitted.

Definition F.9 (Ark Balance)**.** *For a given party P, for a given Ark state $\Sigma = (C, F, S)$ and a given chain view $\mathcal{L}^P_{-k}$, we define the* Ark balance *$b_P^{\Sigma,\mathcal{L}}$ of P at Σ as the sum $b_P^{\Sigma,\mathcal{L}} := \sum_{\texttt{vtxo} \in V_P^\Sigma} \texttt{vtxo.value} + \sum_{\texttt{out} \in R_P^{\mathcal{L}}} \texttt{out.value}$, where V_P is the set of all $\texttt{vtxo} \in C$ which are more than $2k$ blocks from expiry, for which* only *P can provide a witness for a unilateral script path of $\texttt{vtxo}$ and possesses all fully signed transactions in $\texttt{path}(\texttt{vtxo})$, and where $R_P^{\mathcal{L}}$ is the set of all unspent boarding outputs, i.e., all $\texttt{out} \in \mathcal{L}^P_{-k}$ with $\texttt{out.lockScript} = \texttt{Taproot}(\texttt{False}, S_C, S_U)$ where each script path in S_C requires O's signature and where* only *P can provide a witness for the script paths in S_U.*

Theorem F.10 (User Balance Security)**.** *The Ark protocol $\Pi_{\mathsf{ark}}(O)$ with batch expiry time t_e guarantees that for each honest party P, each Ark state $\Sigma = (C, F, S)$, and each chain view $\mathcal{L}^P_{-k}$, P can claim its Ark balance $b_P^{\Sigma,\mathcal{L}}$ onchain, subtracting mining fees, w.o.p.*

Proof. By part (ii) of Theorem F.7, P can turn each $\texttt{vtxo} \in V_P$ into a UTXO confirmed onchain w.o.p., as P is honest and knows $\texttt{path}(\texttt{vtxo})$. Of course, P has to pay the corresponding mining fees to have all these virtual transactions confirmed onchain. Since P also knows the witness of a unilateral script path of each of these newly created UTXOs, as well as the witness of a unilateral script path of each boarding output, P, and only P by part (i) of

Theorem F.7, can spend all these UTXOs (paying mining fees doing so) and thus claim the funds locked by them. Moreover, P can retrieve the funds locked in boarding outputs $\texttt{out} \in R_P^{\mathcal{L}}$ by spending these outputs via one their unilateral script paths (paying mining fees), due to validity of Bitcoin as only P knows corresponding valid witnesses. In conclusion, P can claim its Ark balance $b_P^{\Sigma,\mathcal{L}}$ onchain, up to mining fees, w.o.p. □

Remark F.11. *Note that for the Ark balance to be non-trivial, and for an honest party to actually have retrievable funds, the Ark's batch expiry time should be larger than $2k$.*

Theorem F.12 (Ark Atomicity)**.** *For any Ark state $\Sigma = (C, F, S)$, and for any Ark action $\alpha = (P, N, O, v_{in}, u_{in}, v_{out}, u_{out}, a)$ with $P, O \neq \varnothing$, as long as P, N, or O is honest, $\Pi_{\mathsf{ark}}(O)$ ensures that exactly one of the following will happen:*

(i) α does not get included in a state transition, or

(ii) α is included in a state transition τ resulting in a state (C', F', S'), where:

- $\forall \texttt{vtxo} \in v_{in} : C' \cup F' \not\ni \texttt{vtxo} \wedge S' \ni \texttt{vtxo}$.
- *Each UTXO in u_{in} is spent onchain.*
- $\forall \texttt{vtxo} \in v_{out} : (a = 0 \implies \texttt{vtxo} \in C') \wedge (a = 1 \implies \texttt{vtxo} \in F')$.
- *Each UTXO in u_{out} is confirmed onchain.*

Proof. For an arbitrary Ark state $\Sigma = (C, F, S)$, let us consider an arbitrary Ark action $\alpha = (P, N, O, v_{in}, u_{in}, v_{out}, u_{out}, a)$ with $P, O \neq \varnothing$. Assume that P or O is honest. Then if α is not an Ark action described by the protocol with $P, O \neq \varnothing$, i.e., (a) a boarding, (b) a batch swap, (c) a collaborative exit, or (d) an Ark transaction, the honest participant will not participate in this request, meaning that it will not be included in a state transition. This is because either the honest party is P, and will not provide witnesses approving the action for any of the required inputs from v_{in} and u_{in}, or the honest party is O, and will ignore any action that is not in the form of an Ark transaction or a request to be part of a commitment transaction. Thus, it remains to check that in cases (a)-(d) we end up either with (i) or with (ii):

(a) A boarding action is of the form $\alpha = (P, N, O, \varnothing, \texttt{out}^{\texttt{board}}, v_{out}, \varnothing, 0)$. First, assume that P is honest. In that case, α gives rise to an honestly created boarding request (Routine 2), which does not spend VTXOs that are already spent. This request either gets ignored by O, or it gets (possibly incorrectly) included in a commitment transaction. P will verify this commitment transaction according to Routine 18 (in particular Lines 4-16 and 40), and will only cooperate to create the witness for $\texttt{out}^{\texttt{board}}$ if the verification is successful, according to Routine 22 Line 13. In particular, an honest P will only validate a commitment transaction which spends $\texttt{out}^{\texttt{board}}$ and which has batch outputs that contain all the VTXOs in v_{out}, with the corresponding virtual transaction paths requiring signatures from N. Hence, if this commitment transaction is confirmed onchain, $\texttt{out}^{\texttt{board}}$ will be spent, and each $\texttt{vtxo} \in v_{out}$ will be included in C' (with (C', F', S') being the state after this commitment transaction). If it is not confirmed, α will not get included there, but may be included in a different commitment transaction, for which the same reasoning holds. In conclusion, we either have (i) or (ii).
Now, assume that O is honest. α may only be included in a commitment transaction if O receives the corresponding boarding request and it passes verification according to Routine 3. It will thus only include α in an honestly created commitment transaction, which, if confirmed onchain, guarantees to spend $\texttt{out}^{\texttt{board}}$ and adds all VTXOs in v_{out} to C' of the new state (C', F', S'). Otherwise, if one of the cosigners or requesters is not responding, α will not get included. Note that Line 3 makes sure that the boarding output cannot be double-spent, but even if it could be, this will not lead to atomicity being broken.

(b) A batch swap action is of the form $\alpha = (P, N, O, v_{in}, \varnothing, v_{out}, \varnothing, 0)$. Assume that P is honest. α then gives rise to an honestly created batch swap request (Routine 4), which does not spend VTXOs that have already been spent, and which either gets ignored by O, or incorporated (possibly incorrectly) in a commitment transaction. P will verify this transaction according to Routine 18 Lines 17-31 and Line 40. Only if verification succeeds, will P cooperate with O to create the necessary witnesses for the forfeit transactions spending each $\texttt{vtxo} \in v_{in}$, by Routine 22 Line 23. In particular, an honest P will only cooperate for a commitment transaction which has connector outputs containing anchors for each $\texttt{vtxo} \in v_{in}$, and batch outputs containing each $\texttt{vtxo} \in v_{out}$, with the corresponding virtual transaction paths requiring signatures from N. If this commitment transaction is confirmed onchain, each $\texttt{vtxo} \in v_{in}$ will not be present in C' or F' depending on whether it was the output of an Ark transaction, and will be in S', as the corresponding forfeit transaction can now spend $\texttt{vtxo}$ if ever present onchain, and each $\texttt{vtxo} \in v_{out}$ will be included in the new C'. If not confirmed, α does not change the state. We indeed have either (i) or (ii).
To conclude, assume that O is honest. α is only included in a commitment transaction if O receives the corresponding batch swap request. It will then verify the request according to Routine 8. Consequently, a verified α will only be included in an honestly created commitment transaction. This transaction will only be submitted onchain once O has the fully signed forfeit transactions corresponding to each VTXO in v_{in}. If confirmed onchain, this commitment transaction guarantees that all VTXOs in v_{in} will not be in $C' \cup F'$ and will be in S', and that all VTXOs in v_{out} are in C' of the new state (C', F', S'). Otherwise, if one of the cosigners or requesters is not responding, α will not be included.

(c) We can write a collaborative exit action as $\alpha = (P, \varnothing, O, v_{in}, \varnothing, \varnothing, u_{out}, 0)$. If P is honest, α gives

rise to an honestly created exit request (Routine 5), in particular not spending VTXOs that have already been spent. This either gets ignored by O, or added (possibly incorrectly) to a commitment transaction. P will verify this commitment transaction according to Routine 18 Lines 32-39 and Line 40. P will only cooperate with O to create the necessary witnesses for the forfeit transactions spending each $\texttt{vtxo} \in v_{in}$ if verification succeeds, by Routine 22 Line 31. An honest P will only cooperate for a commitment transaction which has connector outputs containing anchors for each $\texttt{vtxo} \in v_{in}$, and separate outputs corresponding to each $\texttt{out} \in u_{out}$. If this commitment transaction is confirmed onchain, each $\texttt{vtxo} \in v_{in}$ will not be in C' or F' depending on whether it was the output of an Ark transaction, and will simultaneously be added to S', as the corresponding forfeit transaction can now spend $\texttt{vtxo}$ if ever present onchain, and each $\texttt{out} \in u_{out}$ will be available onchain. If not confirmed, α does not change the state. Once again, we have either (i) or (ii).

If we now assume that O is honest, α will only be included in a commitment transaction if O receives the corresponding exit request. It will then verify the request according to Routine 9. Hence, a verified α will only be included in an honestly created commitment transaction. This transaction will be submitted onchain only if O has the fully signed forfeit transactions corresponding to each VTXO in v_{in}. This commitment transaction guarantees that all VTXOs in v_{in} will not be in $C' \cup F'$ but in S', and that all UTXOs in u_{out} are confirmed onchain, once the commitment transaction is confirmed onchain. Otherwise, if one of the cosigners or requesters is not cooperating, α will not be included.

(d) An Ark transaction action can be denoted as $\alpha = (P, \varnothing, O, v_{in}, \varnothing, v_{out}, u_{out}, 1)$. If P is honest, α gives rise to an honestly created Ark transaction request (Routine 7), not spending VTXOs that have already been spent. O either ignores this request, or cooperates with P to provide the required witnesses (possibly after verifying it via Routine 10), making the Ark transaction valid. In the former case, α is not included in a state transition. In the latter case, the existence of valid reset transactions spending each VTXO in v_{in}, implies that these VTXOs are no longer present in C' but instead in S', and the VTXOs in v_{out} are in F'.

Finally, if we assume that O is honest, α will only be included in a state transition if O receives the corresponding Ark transaction request and it passes verification (Routine 10). Only then will O cooperate to provide the required witnesses, leading to the fully signed reset transactions that ensure the VTXOs in v_{in} are not in C' but in S' and the VTXOs in v_{out} are in F'.

□

Theorem F.13 (Operator Balance Security)**.** *The Ark protocol $\Pi_{\mathsf{ark}}(O)$ with batch expiry time t_e and VTXO unilateral delay t_v guarantees that if O is honest and $t_v > 4k$, O can w.o.p. from block height $h + 4k + t_e$ retrieve at least the amount it put in the Ark through commitment transactions confirmed by block height h, subtracting mining fees.*

Proof. We first state and prove a number of intermediate results.

Lemma F.14. *The Ark protocol $\Pi_{\mathsf{ark}}(O)$ with VTXO unilateral delay t_v guarantees, if $t_v > 4k$ and O is honest, that for every Ark state $\Sigma = (C, F, S)$, every $\texttt{vtxo} \in S$ that appears onchain will be spent by a reset transaction or forfeit transaction, w.o.p.*

Proof. Assume without loss of generality that $\texttt{vtxo}$ can be spent by any of its collaborative script paths immediately once posted onchain. Since $\texttt{vtxo} \in S$, an honest operator holds either a valid reset transaction or a valid forfeit transaction (spending an anchor output confirmed onchain) that spends $\texttt{vtxo}$. If the virtual transactions in $\texttt{path}(\texttt{vtxo})$ are submitted to Π_{BTC} at block height t, we know by Corollary that by height $t+2k$, they are in $\mathcal{L}^O_{-k}$. Routine 23 Lines 22-26 instructs the honest operator to post the reset transaction spending $\texttt{vtxo}$ or the forfeit transaction, as soon as $\texttt{vtxo}$ enters O local view of the chain. By height $t + 2k$, $\texttt{vtxo}$ is definitely in $\mathcal{L}^O$, so O will for sure have broadcast the required reset/forfeit transaction(s) by height $t + 2k$. By Corollary F.3, these will be confirmed by height $t + 4k$. By Definition 3.1, any unilateral spend of $\texttt{vtxo}$ is delayed by at least t_v (this is enforced by an honest operator who only includes correctly formed VTXOs in its commitment transactions). Since $t_v > 4k$, there are no possible race conditions and an honest operator is guaranteed to spend $\texttt{vtxo}$ with either the corresponding forfeit or reset transaction. □

Lemma F.15. *Suppose a batch output expires, and the current Ark state is $\Sigma = (C, F, S)$. If $t_v > 4k$, w.o.p., an honest operator can claim all funds locked in the VTXOs contained in the batch that are in $C \cup F \cup S$, i.e., all funds contained in the batch, except those locked in VTXOs that are both unilaterally exited and for which there is no forfeit transaction.*

Proof. By Routine 23 Line 18, an honest operator will perform a sweep operation on the expired batch as defined in Routine 20. If we assume the expired batch output is specified via a VTXT (V, A), every non-leaf transaction in V present onchain will be spent by the operator as the locking script only requires the timelock to be expired and an operator signature. Every leaf transaction has a VTXO locking script, and will therefore not be spent by the operator through the sweep operation. These are exactly the VTXOs that have been unilaterally exited. However, by Routine 23 Line 24 and by Lemma F.14, since $t_v > 4k$, any VTXO that has been spent prior to being put onchain through a unilateral exit, has been spent already w.o.p. by the honest operator via the corresponding reset and/or forfeit transaction, which an honest O has by Theorem F.12. In case of a reset transaction, any of the VTXOs in the subsequent Ark transaction (if valid) that have been spent can also be claimed via a forfeit transaction an honest operator has. In aggregate, this means that the operator claims all funds

locked in VTXOs that either have not been spent and not unilaterally exited (which are the VTXOs of the expired batch that are in $C \cup F$), or have been spent (which are the VTXOs in S, regardless of whether they have been unilaterally exited). That is, an honest operator can claim all funds held in the expired batch, up to those funds that have been unilaterally exited and that have not been spent, or spent in an Ark transaction included onchain. □

Lemma F.16. *Suppose that from block height h onward, an honest operator O does no longer accept incoming batch swap and boarding request, only accepting exit requests, in order to shut down its Ark. Then $\Pi_{\mathsf{ark}}(O)$ with batch expiry time t_e and VTXO unilateral delay t_v guarantees w.o.p. that, if $t_v > 4k$, by block height $h + 4k + t_e$, O has retrieved exactly the value it put into the Ark by funding commitment transactions, subtracting mining fees, and adding at least fees it collected for processing requests.*

Proof. This proof amounts to keeping track of all the flows of funds, in particular, which funds are and will come under control of the operator, assuming this operator exactly follows the protocol. For simplicity, we assume the connector outputs hold a negligible amount of funds. All groups of funds are greater than or equal to zero.

Fix the block height h. We denote the commitment transactions from the operator O that are confirmed onchain by $\mathtt{tx}_j$, where j runs through $0, \ldots, k_h, \ldots, k_e, \ldots$. The index k_h refers to the latest commitment transaction confirmed before or at block height h, and k_e to the latest commitment transaction confirmed before or at block height $h + 2k + t_e$. Furthermore, we denote by $e(j)$ the latest commitment transaction before the expiry of the batches in $\mathtt{tx}_j$. For example, we have $k_e = e(k_h)$.

By construction of any honestly created commitment transaction $\mathtt{tx}_j$ as specified in Routine 13, we can categorise the following incoming and outgoing groups of funds, based on the transaction inputs and outputs:

- Incoming funds:
  - L_j: the liquidity coming directly from O to finance $\mathtt{tx}_j$.
  - B_j: the funds coming from boarding transactions.
- Outgoing funds:
  - V_j: the funds locked in batch outputs holding VTXOs.
  - U_j: the funds locked in UTXOs as a consequence of exit requests.
  - M_j: the mining fees.

We can further categorise the outgoing funds V_j and U_j by tracking which VTXOs or boarding transaction outputs were used as part of the boarding, batch swap, or exit requests that led to the formation of the VTXOs and UTXOs in $\mathtt{tx}_j$. That is, we can write $V_j = \sum_{i<j} V_{i,j}$, where $V_{i,j}$ is the aggregate value of all VTXOs or boarding transaction outputs that were spent in order to (partially) fund the VTXOs in $\mathtt{tx}_j$. Notice that $V_{j,j} = B_j$, as all the boarding transaction outputs spent in $\mathtt{tx}_j$ are used to create new VTXOs in $\mathtt{tx}^j$ by Routine 13, and O ensures to not have more value locked in the VTXOs created from a boarding request than the value locked in the corresponding boarding output (Routine 3 Line 4). Similarly, we write $U_j = \sum_{i<j} U_{i,j}$ (notice the strict inequality now).

Since any commitment transaction must be a valid Bitcoin transaction, we must have the following conservation of funds, for every $j \geq 0$:

$$L_j + B_j = V_j + U_j + M_j.$$

Recalling that $B_j = V_{j,j}$, we can rewrite the above to:

$$L_j = \sum_{i<j} V_{i,j} + \sum_{i<j} U_{i,j} + M_j. \tag{2}$$

We can also establish a second conservation identity, realising that with an honest operator, a VTXO is either spent as part of a batch swap or exit request, leading to a VTXO or UTXO in a later commitment transaction, or through a unilateral exit, or not spent at all. Note that a VTXO can also be spent by an Ark (reset) transaction. However, the funds in that VTXO will just be relocated in another VTXO, that will either be batch swapped, exited, or not spent at all. Since an honest operator does not allow for spending a VTXO that has already been spent in an Ark transaction (via a reset transaction) (by Routines 8 Lines 1-4, 9 Lines 1-4, 10) Lines 1-5), we can just distribute these funds over the other categories. Hence, for every $j \geq 0$, and because an honest operator will only let a VTXO be spent exactly once (again, by Routines 8 Lines 1-4, 9 Lines 1-4, 10 Lines 1-5), and not allow more funds in the outputs than in the inputs of each batch swap or exit (by Routines 8 Line 8 and 9 Line 5), we have:

$$\sum_{i \leq j} V_{i,j} = \sum_{k>j} V_{j,k} + \sum_{k>j} U_{j,k} + E_j^U + X_j + F_j. \tag{3}$$

We introduced the term E_j^U to indicate unilateral exits of *unspent* VTXOs or VTXOs spent by Ark transactions (via reset transactions). We explicitly do not account for spent VTXOs that one attempts to unilaterally exit with as well, as this would result in double counting. Also, we introduced X_j to account for all VTXOs that have not been spent. Finally, F_j represents the fees that the operator may charge for processing requests. Indeed, when creating the new outputs in return for batch swap/exits, their aggregate value might be less than V_j, where the difference has been claimed by the O once the batch expires.

By Lemma F.15, since $t_v > 4k$, an honest operator will be able to claim w.o.p. all the funds held in expired batches, except for the unilaterally exited VTXOs for which O does not hold a forfeit transaction. In other words, the amount S_j^O swept by the operator is given by:

$$S_j^O = \sum_{i \leq j} V_{i,j} - E_j^U - E_j^M - S_j^M, \tag{4}$$

where the last two terms E_j^M are the funds the operator needs to give in mining fees for forfeit transactions, Ark (reset) transactions spending unilaterally exited, spent VTXOs (Routine 23 Lines 22-26), and S_j^M are the funds the

operator needs to give in mining fees for sweep transactions, respectively.

Now, realise that since O does not accept boarding or batch swap requests after block height h, $S_j^O = 0$ for $j > k_h$. Indeed, $(\mathtt{tx}_j)_{j>k_h}$ will have $B_j = V_j = 0$. Only U_j may be non-zero, containing UTXOs from users exiting collaboratively from batches with index less than or equal to k_h. More specifically, $U_{i,j}$ may be non-zero for $i \leq k_h$, but $U_{i,j} = 0$ for $i > k_h$. Also, realise that $U_{i,j} = 0$ for $j > k_e$, as by then all batches confirmed before or at height h have now expired and an honest operator would not perform a collaborative exit for a VTXO coming from an expired batch.

We can thus focus on the commitment transactions confirmed before or at block height h. By block height $h+2k+t_e$, every batch contained in one of the commitment transaction $(\mathtt{tx}_j)_{j=0}^{k_h}$ will have expired due to Routine 13, and by Routine 23 an honest operator will have initiated sweeps for all expired, unspent funds. The latest sweep transactions will thus be submitted at height $h + 2k + t_e$, and will be confirmed by height $h + 4k + t_e$ w.o.p. by Corollary F.3. Note that as long as a sweep is not confirmed onchain, a user might still try to unilaterally exit funds. In case the user's exit transactions get confirmed instead of O's sweep transactions, we count the corresponding funds as a part of E_j^U. The total amount gained through commitment transactions before and at block height h is thus:

$$\begin{aligned}\sum_{j\leq k_h} S_j^O &\overset{(4)}{=} \sum_{j\leq k_h}\Big(\sum_{i\leq j} V_{i,j} - E_j^U - E_j^M - S_j^M\Big)\\ &\overset{(3)}{=} \sum_{j\leq k_h}\sum_{k>j}(V_{j,k}+U_{j,k})\\ &\quad + \sum_{j\leq k_h}(X_j + F_j - E_j^M - S_j^M), \qquad (5)\end{aligned}$$

Since $V_k = 0$ for $k > k_h$, we rewrite the first term of (5) as

$$\begin{aligned}&\sum_{j\leq k_h}\sum_{k>j}(V_{j,k}+U_{j,k})\\ &\quad= \sum_{j\leq k_h}\sum_{j<k\leq k_h}(V_{j,k}+U_{j,k}) + \sum_{j\leq k_h}\sum_{k>k_h} U_{j,k}\\ &\quad= \sum_{k\leq k_h}\sum_{j<k}(V_{j,k}+U_{j,k}) + \sum_{j\leq k_h}\sum_{k>k_h} U_{j,k}\\ &\quad= \sum_{k\leq k_h}\sum_{j<k}(V_{j,k}+U_{j,k}) + \sum_{k_h<k\leq k_e}\sum_{j<k} U_{j,k}, \qquad (6)\end{aligned}$$

where the second equality is just a rearrangement of the sums, and the third equality first uses that $U_{j,k} = 0$ for $j > k_h$ and for $k > k_e$, and then rearranges the sums.

On the other hand, the funds put in by the operator through commitment transactions before and at block height h add up to

$$\begin{aligned}\sum_{j\leq k_e} L_j &\overset{(2)}{=} \sum_{j\leq k_e}\sum_{i<j}(V_{i,j}+U_{i,j}) + \sum_{j\leq k_e} M_j\\ &= \sum_{j\leq k_h}\sum_{i<j}(V_{i,j}+U_{i,j}) + \sum_{k_h<j\leq k_e}\sum_{i<j} U_{i,j}\\ &\quad + \sum_{j\leq k_e} M_j, \qquad (7)\end{aligned}$$

where in the second equality, we used that $V_j = 0$ for $j > k_h$. Substituting (6) into (5), and then combining the latter with (7), we obtain the net balance of the operator at block height $h + 4k + t_e$:

$$\sum_{j\leq k_e} S_j^O - \sum_{j\leq k_e} L_j = \sum_{j\leq k_h}(X_j + F_j - E_j^M - S_j^M) - \sum_{j\leq k_e} M_j,$$

which means that O retrieves all funds that it put into the Ark, subtracting the mining fees and adding the fees the operator charges for processing requests. Additionally, the operator may gain any funds from VTXOs that have not been spent in any way before their batch expired, and have therefore just been swept by the operator. This identity then immediately gives us a necessary condition for running an Ark profitably; the operator fees should cover the mining fees (minus any potential unspent VTXOs that have been swept). □

Lemma F.16 now finishes the proof of Theorem F.13. Indeed, if an operator wants to retrieve its funds it put in up to and including block height h, it can always decide to stop processing boarding and batch swap requests received after h, and be guaranteed w.o.p., if $t_v > 4k$, to have retrieved from block height $h + 4k + t_e$ onward at least the funds it put in via commitment transactions confirmed by block height h, subtracting mining fees. □